Towards Further Formal Foundation of Web Security: A Security Model with Cache by Expression of Temporal Logic in Alloy

Hayato Shimamoto, *Non-Member,* Naoto Yanai, *Non-Member,* Shingo Okamura, *Member, IEEE,* Jason Paul Cruz, *Member, IEEE,* Shouei Ou, *Non-Member*

***Abstract*—Security analysis of a web system is complicated, and thus analysis using formal methods to describe a system specification mathematically has attracted attention. Previous studies that use this kind of approach do not consider a cache, which is an essential function for the web. Meanwhile, several attacks using caches, such as extracting account pages of users in social network services, have been introduced. In this paper, we propose a web security model that includes a cache and show that our model can express attacks, such as unauthorized access to users' account pages via the cache.**

A *temporal logic*, which expresses time series, is necessary for the expression of a cache. However, introducing temporal logic to current web security models is not trivial. Specifically, Alloy that is used in implementations of current models does not have the ability to express temporal logic. Although current models have shown several concepts of time series, the temporal logic they use is insufficient because it does not consider parallel communication.

Therefore, in this paper, we first devise an expression of temporal logic in Alloy that can express state transitions of elements in the web for the underlying purpose, i.e., a web security model that includes a cache. The improvements in the expressiveness of the proposed model are shown via several examples, such as basic behaviors of a cache and browser cache poisoning attack. Our source code in Alloy is publicly available.

***Index Terms*—Security Analysis, Web Security, Formal Methods, Security Model, Temporal Logic, Cache**

I. Introduction

A. Background

The web is used for many modern services and systems, such as social network services (SNS) and content delivery. The web is a complicated system composed of various elements and protocols, and its wide use makes it prone to attacks. For example, given that current SNS contain cross-domain systems that provide services via multiple domains, malicious contents may be injected and sent within the cross-domain systems [1], [2]. Moreover, an attack [3] wherein an account page of a client is maliciously generated on a server and accessed by an adversary was discovered recently. Therefore, the web requires strict security.

Two methods are used in security analysis of a web system. The first method is the simulation of the behavior of a system. This security analysis method has relatively low costs and is therefore generally used in the industry. However, the structure of the web has become complicated and contains a variety of elements. Consequently, security analysis that simulates a complicated web structure generates a leakage and therefore cannot guarantee security rigorously. The second method is a *formal method* that describes a security model to express a system specification in proposition logic. In this method, the security of a system is analyzed mathematically without any leakage as long as the security model can express a system precisely. Ahkawe et al., [4] proposed a basic model that supports the security analysis of subsequent web research, and their work has been extended by De Ryck et al. [5] and other works [6]–[10].

Although expressing a system exactly as a security model is significantly important in the formal method, we consider that the model of Ahkawe et al. has several insufficiencies. First, **their model does not include a cache**, which is a kernel function widely used in the web for storing and reloading contents. Consequently, their model cannot analyze certain risks, such as attacks utilizing a cache and its stored contents. The attacks described above [1]–[3] utilize a cache, and thus a security model considering behavior of a cache is necessary. Second, the expressiveness of *temporal logic* in their model is insufficient because **it does not consider parallel communication**. Temporal logic is a proposition logic that expresses a change in time and is utilized in a security model to express state transitions within a system. A model with insufficient temporal logic expressiveness cannot be used to analyze the web because it cannot express state transitions within a system. In particular, this kind of model cannot analyze a situation where multiple sessions are executed in parallel. Handling multiple sessions in parallel is crucial in services, such as SNS, where multiple users access the same service or a single user utilizes several services simultaneously. This situtation is similar to introducing a cache. That is, the temporal logic problem remains unsolved in subsequent works, including the work of De Ryck et al. [5]. Therefore, it is important to extend previous results to promote further analysis support in future research.

B. Contribution

In this paper, we propose a web security model that includes a cache. We then show that the expressiveness of the proposed

H. Shimamoto, N. Yanai, J. P. Cruz, and S. Ou are with Osaka University, Japan.

S. Okamura is with National Institute of Technology, Nara College, Japan.

Manuscript received July 1, 2018.

model is stronger than those of current models by verifying basic behavior of a cache and related attacks, such as browser cache poisoning attack [2] and web cache deception attack [3], as case studies. By analyzing these attacks, existence of vulnerability about malicious access to user information in web systems such as SNS can be verified.

Our primary contribution is the proposal of a new syntax of the temporal logic in Alloy that can express state transitions of elements in the web. Our syntax solves the aforementioned problem of temporal logic in the previous section and can potentially enable various analyses following the research of Akhawe et al. [4] (See Sections III and IV for details). Therefore, the verification of a situation where multiple HTTP sessions are carried out in parallel, including related attacks that manipulate contents, becomes available.

The problem this paper solves is described below. The expression of current models is insufficient because Alloy does not have a syntax that expresses temporal logic. Expression of temporal logic in Alloy is not trivial. Some models [4], [5] expressed temporal logic by proposing new syntaxes, but these syntaxes have strict expressiveness restrictions and are unsuitable for general analysis of the web (See Section II for details). Intuitively, this problem can be solved by improving the expressiveness of the temporal logic in Alloy. We published our source codes in Alloy on GitHub (**https://github.com/sho-rong/webmodel**).

C. Related Works

1) Formal Methods for the Web: Many studies investigated formal methods for web security, as summarized in [11], but only the work of Ahkawe et al. [4] performs verification of the web as a platform. Similar to the models of Ahkawe et al. [4] and De Ryck et al. [5], Chen et al. [12] verified app isolation using multiple browsers, but they did not mention the problem described in this paper. Bansal et al. [13], [14] modeled a basic element of the web, such as a browser or a server, in ProVerif. Fett et al. [15]–[18] proved the security for a single sign-on (SSO) system. These works are similar to our work in the sense that the web is modeled and verified in platform levels, but they do not consider a cache mechanism and they use different utilization tools and approaches.

We now describe several case studies that verified technologies used in the web. Chaitanya et al. [6] verified the validity of CORP, which is one of the security requirements in the web. Chen et al. [8] proposed ASPIRE, a framework of web applications, and verified its security when its behavior is incorporated in the web. Somorovsky et al. [10] performed security analysis of cloud computing in the web and discussed countermeasures for a discovered vulnerability. Our work is different in the sense of handling fundamental functions of the web.

2) Formal Methods in Alloy: The following works used Alloy. Klein et al. [19] analyzed an unexpected behavior of micro-kernel of an OS called seL4 and proved that such behavior was impossible. Shin et al. [20] verified the security of the Android OS. Lie et al. [21] inspected the presence of a state where a processor loses its tamper resistance and discovered the condition falling into the state. Near et al. [22] discovered access patterns that satisfy the security in the web application including some specific elements. These works are different from our work because we abstract specifications of the web as a platform and not individual techniques.

3) Web Security: The current specification of HTTP has various vulnerabilities. Jia et al. [2] used a browser cache to discover browser cache poisoning attack, which allows the target browser to behave arbitrarily by storing forged contents. De Ryck et al. [5] used cookies in a browser as a countermeasure for an attack called cross-site request forgery. Ogawa et al. [3] used a cache of an intermediary, such as a proxy, to discover web cache deception attack, which allows an attacker to extract a file it has no permission to access. We will discuss these attacks in Section VI to show the advantages of our proposed model.

Several vulnerabilities of HTTPS, an extension of HTTP, have been discovered [23]. Consequently, HTTP strict transport security (HSTS) [24] and public key pinning extension for HTTP (HPKP) [25], which are extensions that improve the security of HTTPS, have been developed. Most current implementations cannot preserve the security of these protocols because their server setup is insufficient [26]. We consider that these protocols should be analyzed to establish a standard setting that can guarantee sufficient security in the future. We consider that our proposed model can be used in the analyses of HSTS and HPKP, and we leave these analyses as an open problem.

4) State-of-the-art Formal Methods: In recent years, Z3 [27] has been commonly used in analysis using formal methods [28]–[30]. Bocić et al. [31] used Z3 in web-related analysis and showed the model extraction for dynamically typed script languages. However, Z3 is based on command lines and its outputs are in text format, and thus analysis of results obtained from complicated models, such as the web, needs significant effort. On the other hand, Alloy Analyzer is suitable for analysis of the web because its output is in the form of a graph chart, which is identical to actual communication.

D. Paper Organization

The rest of this paper is organized as follows. First, we provide a background required for understanding this paper in Section II. Then, we describe current models and their temporal logic problems in Section III and then show our syntax that solves these problems in Section IV. Then, we propose the model including a cache in Section V and describe case studies in Section VI. Finally, we present the conclusion and future direction in Section VII.

II. Preliminaries

In this section, we present backgrounds of formal methods, the web platform, and the hypertext transfer protocol that are necessary to understand this paper.

A. Formal Methods

1) Overview: Formal methods using mathematical verifications have been used in security analysis of systems in general. In a formal method, a user makes a security model for a target system and checks this model to confirm the security of the system. A formal method called theorem prover performs mathematical proofs through interactions with a user, but we discuss model checking as a main approach in this paper.

A user performs the following procedure when utilizing a formal method for system development. First, the user prepares a security model for the target system. Then, the user carries out model checking for the security model. Then, if the user discovers insufficiencies in specifications and weaknesses of the system based on the output, then the user devises corresponding countermeasures. Finally, the user revises the security model according to the countermeasures and performs the procedure again. The procedure is repeated until no weaknesses are found.

2) Security Model: A security model expresses a target system as the use of a proposition logic [32]. A security model describes the structures and behaviors of the target system, the threat model of the system (e.g., ability of attackers), and the security requirements of the system.

3) Alloy Analyzer: Alloy Analyzer is a model-checking tool that uses a formal method. In this analyzer, a user describes the security model of a target system with a language called Alloy. A model in Alloy can have a syntax of unbounded size, and the syntax is instantiated by specifying a size bound when the model is executed. In an actual analysis, an Alloy code as abstraction of a system specification is converted into a satisfiability (SAT) problem, which is then solved by a SAT solver within the Alloy Analyzer. The output can be either of two states of the model, i.e., one that satisfies the conditions and one that does not. The former output is used in a case study to confirm that the security model is implemented properly. The latter output is used for security analysis of the system, where the behavior of the system that does not satisfy security requirements is output by describing the security that the system should satisfy. This behavior includes a vulnerability that violates security requirements, and thus a user can discover the vulnerability by analyzing the output. The Alloy Analyzer is intuitive to use because the outputs are in the form of a graph, which is not supported in other tools that use formal methods.

We briefly explain the Alloy language used in the Alloy Analyzer. In Alloy, all data types are represented as relations and are defined by their *type signatures*. A type signature declaration consists of the type name, the declaration of *fields*, and an optional *signature fact* constraining elements of the signature. A *subsignature* is a type signature that extends another as a subset of the base signature, and an *abstract signature* represents a classification of elements that is aimed to be refined by a more concrete subsignature.

B. Web Platform

The web is a system that provides documents on the Internet with HyperText described in HTML. The HyperText can express relationships between documents by including their links. The links in the web create a connection of various documents on the Internet and facilitate the downloading of these documents. This convenient feature has made the web indispensable in the current society.

C. Hypertext Transfer Protocol (HTTP)

Hypertext transfer protocol (HTTP) is a protocol that realizes general communication for various types of data. While there are various versions of HTTP, we refer to HTTP/1.1 [33]–[38], which is commonly used and has HTTP/1.0 [39] backward compatibility.

HTTP programs include various roles that are classified into three entities, namely, client, server, and intermediary. These programs may play multiple roles simultaneously, i.e., a program may be a client for a connection as well as a server for another connection. A client in the HTTP establishes a connection to a server to send a request, and this role is generally performed by a web browser. A server in the HTTP sends a result to the client as a response to the received request. Several servers and clients communicate through multiple programs, which are called intermediaries. HTTP/1.1 has three kinds of intermediaries, namely, proxy, gateway, and tunnel. In this paper, we only target proxy and gateway as programs that include a cache. A proxy can edit a request and a response, convert communication contents into some specified format, and delete the contents. A gateway connects a local network and a global network, where it receives a request from the global network side as a server. The gateway also sends a request to the most suitable server among multiple servers in a local network as a client. The gateway generates a response based on the results obtained from the server in the local network and responds to the origin of the original request in the global network.

A basic communication in HTTP consists of two phases, a request from a client to a server and a response from a server to a client. At the beginning of the phases, a client sends a request that points to a target resource that a server owns. The server then transmits a response that includes the target resource to the client after receiving the request. Using the procedure above, a client can acquire a resource. The underlying purpose of HTTP is a protocol for general usage, and thus contents defined on a protocol include semantic structures of communication contents, usage purpose of communication contents, and behavior of a communication partner. A sender generates and sends a packet along with the defined semantics, and the receiver behaves along with the packet. Using this simple design, the usage to relay the communication between protocols different from HTTP is enabled.

III. Current Web Security Models and Their Temporal Logics

In this section, we describe two web security models, including their expressiveness and problems in their temporal logics. A temporal logic pertains to an expansion of a proposition logic such that the logic can express state transitions along with time series. We first recall three types of a threat model shown in the previous work [4]. Then, we show the models of Ahkawe et al. [4] and De Ryck et al. [5] and their temporal logics. Finally, we describe the problems with these temporal logics.

A. Threat Model

In this paper, an attacker is classified into three types: *web attacker*, *network attacker*, and *gadget attacker*. The web attacker, who is the most basic attacker among the three types, has root authority in at least one web server and can generate a response to any content of the request to the server. The web attacker possesses multiple domain name system (DNS) servers and obtains a server certificate for a domain it owns from an authorized certificate authority. The web attacker can respond to a request to the server it manages and send a request to a server that is managed by a legitimate user via a terminal it owns. We note that requests from the web attacker do not need to follow the specification of HTTP. Moreover, the web attacker can use API of a browser arbitrarily when a browser accesses the website owned by the web attacker even once. However, API used by the attacker cannot behave beyond the security policy set by the browser.

The network attacker can eavesdrop and falsify contents of unencrypted communication (e.g., HTTP communication) and interrupt communications, but it cannot intervene in HTTPS. However, the network attacker can issue a self-signed certificate when it has obtained a server certificate for a malicious DNS it owns from a legitimate certificate authority. Then, the network attacker can use this self-signed certificate to intercept and intervene in HTTPS communication.

The gadget attacker has the abilities of both the web and network attackers and it can insert contents of several specified formats in legitimate websites. The formats of these contents depend on web applications, and hyperlinks can be injected in many situations.

Similarly, behavior of legitimate users has restrictions. A legitimate user refers to a general user who is different from the three types of attackers described above. When the behavior of legitimate users is unrestricted, a very simple behavior that violates the security such that a legitimate user may send a password to the attacker will be detected, and the number of output results becomes enormous. The restrictions for legitimate users then reduce the number of the output results, and push forward verification smoothly. An appropriate restrictions adjustment is important because typical attacks are overlooked when excessively strong restrictions are considered.

In the basic model by Ahkawe et al., the following restrictions are followed by legitimate users. First, a user may be connected to multiple websites, including that owned by an attacker. Intentional connection by the user to the malicious site is not included. Next, a user never confuses malicious websites from legitimate websites even if the user connects to a malicious website. This premises that a user understands security alerts of browsers. Under these assumptions, the following two conditions are defined as security requirements of the web. The first condition is security invariants– specifications of web components are not modified, i.e., the condition requires that the elements follow specifications. The second condition is session integrity– the server managed by a legitimate user replies only to an HTTP request generated by a legitimate user.

B. Basic Model

The web security model proposed by Akhawe et al. [4] is a fundamental model that aims to have extensibility. This model selects and contains components that are used frequently in the web. This model implements a temporal logic expressing the temporal axis to express the order of events, such as a request and a response. We describe the `Time` class as the temporal axis, as shown in Code 1.

Code 1
TEMPORAL AXIS IN THE BASIC MODEL

```
open util/ordering[Time]

sig Time {}

fact Traces{
  all t:Time- last | one e:Event | e.
        pre=t and e.post=t.next
  all e:Event | e.post=e.pre.next
}
```

Ordering of instances of the `Time` class is enabled by an `ordering` option, and operators, such as `next` expressing the next instance, `first` expressing the first instance, and `last` expressing the last instance, are available. We can express the order of events via the temporal axis of the `Time` class by associating the `Event` class expressing a request and a response with the `Time` class.

C. Cookie Model

De Ryck et al. [5] proposed a web security model that pointed out the lack of inclusion contents of the basic model (similar to our work), and they created an extension that includes cookies. We call their model as the Cookie model hereinafter. The Cookie model applies the temporal axis of the event introduced in the basic model and extends the temporal logic to express state transitions of cookies at the time of event occurrence. The code for the temporal logic of the Cookie model is shown in Code 2.

Code 2
TEMPORAL LOGIC OF THE COOKIE MODEL

```
sig CSState {
  dst: Origin,
  cookies: set Cookie
}

sig CSStateHTTPTransaction extends
      HTTPTransaction {
  beforeState: CSState,
  afterState: CSState
```

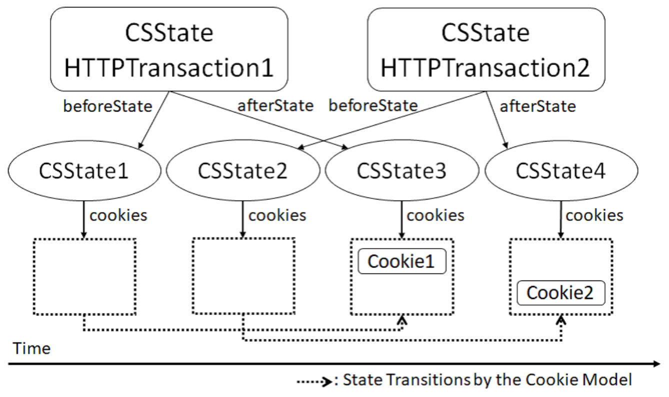

Fig. 1. An Example of State Transitions Available in the Cookie Model

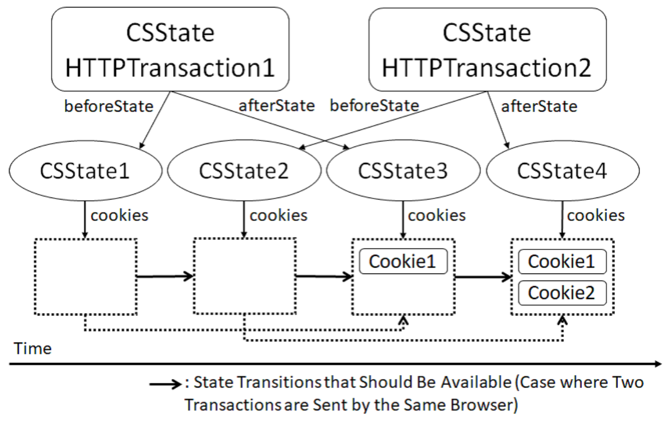

Fig. 2. An Example of State Transitions Unavailable in the Cookie Model

```
}{
    beforeState.dst = afterState.dst
    afterState.cookies = beforeState.
          cookies + (resp.headers &
          SetCookieHeader).thecookie
    beforeState.dst = req.host
}
```

We define the `CSState` class to express a state of a cookie at each time. We can express a state of a cookie in a browser at the time of a request and a response by associating the `CSState` class with the `HTTPTransaction` class expressing a request and a response. Moreover, the `CSState` class represents how a set of cookies changes between states of a request and a response. Therefore, state transition of cookies between a request and a response can be expressed using Code 2.

D. *Problems with Current Models*

The temporal logics of the models described in Sections III-B and III-C have problems with their expressiveness. In particular, given the ability of these temporal logics, state transitions can be expressed only between a single request and a single response. In other words, state transitions with more than two states cannot be expressed. For example, consider the situation where two requests are sent in succession by the same browser, and responses are sent in turn, as shown in Fig. 1. The expressiveness in the Cookie model can express changes in the set of cookies, such as CSState1 to CSState3 and CSState2 to CSState4. However, the expressiveness in this model cannot store Cookie1 stored in CSState3 into CSState4 because CSState4 cannot catch state transitions that occurred in CSState3, as shown in Fig. 1. However, a state shown in Fig. 2 should be expressed when a user considers the behavior of cookies. In this paper, we propose a syntax in Alloy that realizes state transitions along the temporal axis towards a situation shown in Fig. 2.

IV. Proposed Syntax of Temporal Logic in Alloy

In this section, we propose a new syntax of temporal logic in Alloy that can express state transitions of various elements in the web at each event occurrence. First, we explain our main idea that aims to solve the problems described in the previous section. Then, we describe the expression of the temporal axis defined in Section III-B and our general state class. Our general idea is to consider two predicates, one that decides the last state and another one that decides the initial state.

A. *Main Idea*

The main problem of the expressiveness of temporal logics in current models is that **state transtitions with more than two states cannot be expressed**. This problem is caused by the relationship between state classes, i.e., the `CSState` class in the Cookie model is expressed using only the `HTTPTransaction` class. Such method can only express state transitions in the same `HTTPTransaction`. Therefore, a method that can express the relationship between state classes without depending on the `HTTPTransaction` class is necessary. In addition, the `CSState` class in the Cookie model that expresses the state of a cookie lacks extensibility to express other elements in the web. Therefore, we aim to express the relationship between state classes to strengthen extensibility and define a general state class that can express states of various elements in the web.

In this paper, we consider temporal logic, which is necessary for expressing state transitions between state classes. We consider the kind of predicate that is necessary to express transitions through the whole temporal axis. If we can determine whether two states are successive on the temporal axis, then we will be able to express **the anteroposterior change that is possible in two states**. In addition, we will be able to express **a condition on an initial state, i.e., an initial condition,** if we can decide the initial state of state transitions. By combining these two expressiveness, we can then express possible state transitions in the whole temporal axis inductively. Based on the idea described above, we construct predicates that decide the initial state and last state in transitions to express state transitions in the whole temporal axis.

B. *Expression of Temporal Axis*

In the basic model [4], the `Event` class expressing a response and a request on a network is associated with the

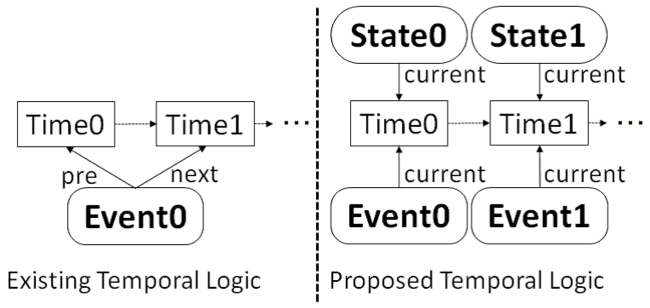

Fig. 3. Relation between Temporal Axis and Event in the proposed model

`Time` class expressing the temporal axis. The relationship between these classes can represent contents, e.g., Event0 is produced between Time0 and Time1 in Fig. 3, and can also express the time of an event through an instance of two `Time` classes. Such relationship is available because only the `Event` class is associated with the temporal axis.

In the proposed syntax, not only the `Event` class but also the `State` class expressing states of elements on the web at a certain point in time are related to the temporal axis. We emphasize that the current model expresses the time of one event by instances of two `Time` classes, and thus the logical expression of the predicate and its implementation become complicated. Therefore, we construct a syntax such that the time of events can be expressed in a single instance of a `Time` class, as shown in Fig. 3.

The basic model includes the restriction such that a request and a response are not created at the same time, and our proposed syntax removes this restriction. We note that our proposed syntax does not affect the extensibility of current models, i.e., the expressiveness of their temporal logics remains unchanged.

C. General State Class

We define a new `State` class as shown in Code 3. The flow of the `State` class in line 2 connects `State` classes and represents state transitions, and `current` represents the time when the states can be obtained. Moreover, we also define the `EqItem` class in line 7 and `DifItem` class in line 8 as variables of the `State` class.

Code 3
OUR PROPOSED STATE CLASS

```
1  abstract sig State{
2    flow: set State,
3    eq: one EqItem,
4    dif: one DifItem,
5    current: set Time
6  }
7  abstract sig EqItem{}
8  abstract sig DifItem{}
9  sig StateTransaction extends
        HTTPTransaction{
10   beforeState: set State,
11   afterState: set State
12 }
```

The `EqItem` class expresses an element in the web that remains the same even after state transition, i.e., *invariants*. We use these invariants in the presence of multiple states to determine which states are on the same transition. The web contains various elements and state transitions occur in parallel, and thus deciding which instances belong to the same transition is necessary. Such decision is enabled by invariants. The `DifItem` class expresses an element in the web that may change during state transitions, i.e., *variants*. The `beforeState` and `afterState` are defined to express the time of a request and a response, respectively, by relating the states with `HTTPTransaction` similar to the Cookie model. We define the dedicated class for elements extended from `State`, `EqItem`, and `DifItem` classes, which are used to consider state transitions of elements in the web. We also clarify invariants and variants of the elements and describe them in the extended class.

Code 4 refers to our application to cookies based on the Cookie model, where `CookieEqItem` and `CookieDifItem` have the same functionality as `EqItem` and `DifItem` in Code 3, respectively. In the Cookie model, a client is an unchangeable item, i.e., an invariant, and the set of cookies is a changeable item, i.e., a variant. Each client saves cookies that can change during state transition and the client is not changed during state transition.

Code 4
APPLICATION TO COOKIES

```
sig CookieState extends State{}{
  eq in CookieEqItem
  dif in CookieDifItem
}
sig CookieEqItem extends EqItem{
  client: one HTTPClient
}
sig CookieDifItem extends DifItem{
  cookie: set Cookie
}
```

D. Predicate that Decides the Last State

For the `State` class described in IV-C, we define a predicate named `LastState` that decides whether a state is on the same state transitions. `LastState` takes three parameters, two of which are states. We call the two states as `pre` and `post` and are defined in the first line of Code 5. We define the predicate `LastState` as follows: a `pre` state is the previous state of a given `post` state only if the `LastState` is true. However, when only these states are utilized as input, a single `post` state may have multiple `pre` states. To avoid this, we further introduce `StateTransaction` to specify the time for each `post`. We define such a parameter (`StateTransaction`) as `str` in the first line of Code 5. With `str`, we can decide which `pre` is the previous state for the `post` based on the time included in the given `StateTransaction`. That is, we can uniquely decide a pair of a `pre` and a `post`, whose predicate becomes true.

Code 5
PREDICATE THAT DECIDES THE LAST STATE IN STATE TRANSITIONS

```
pred LastState[pre:State, post:State,
      str:StateTransaction]{
  pre.eq = post.eq
  post in str.(beforeState +
         afterState)

  some t,t':Time |
    {
      t in pre.current
      t' in str.(request + response +
             re_res).current
      t' in str.request.current
            implies post in str.
            beforeState
      t' in str.(response + re_res).
            current implies post in
            str.afterState
      t' in t.next.*next

      all s:State, t'':Time |
        (s.eq = pre.eq and t'' in s.
             current) implies
          (t in t''.*next) or (t''
                 in t'.*next)
    }
}
```

We then define a condition where a predicate `LastState` is true as follows:

Definition 1 (LastState): Suppose `pre` and `post` are instances in the `State` class, and `str` is an instance in the `StateTransaction` class. We say `LastState` is true if a `pre`, a `post`, and a `str` meet the followings conditions:

- invariants of the `post` are identical to that of the `pre`;
- the `post` belongs to either `beforeState` or `afterState` for the `str`;
- there are no other states whose invariants are identical to the given `pre` and `post` states;
- and the time of `str` is between the time of the `pre` and time of the `post`.

E. Predicate that Decides the Initial State

The predicate named `InitialState` is available for a `State` class and decides an initial state, as shown in in Code 6. `InitialState` takes `State` class as input, refered to as `s` in the first line, and decides whether the state is an initial state. Here, there are multiple initial states if invariants are different from each other because state transitions become independent of each other, as shown in Fig. 4.

Code 6
PREDICATE THAT DECIDES THE INITIAL STATE ON STATE TRANSITIONS

```
pred InitialState[s:State]{
  all s':State |
    s.eq = s'.eq implies
          s'.current in s.current.*next
}
```

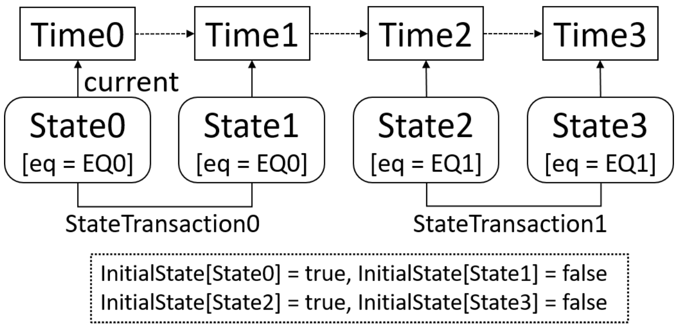

Fig. 4. Example of InitialState

Definition 2 (InitialState): We say, for any instance `s` in a `State` class, `InitialState` is true with respect to `s` if all instances in the `State` class, whose invariants are identical to that of `s`, occur only after `s` occurs.

V. PROPOSED MODEL

In this section, we propose a web security model that includes a cache mechanism as an application of the proposed syntax described in Section IV. The cache is an important element of the web and has therefore been utilized in many attacks as described in Section I. Vulnerability utilizing a cache has significant influence to web users, and therefore analyzing the security of both the web and caches is crucial.

A. Features of the Proposed Model

We construct the proposed model.

1) Cache: A cache belongs to a client, a server, and an intermediary, and its basic behaviors include *storage*, *deletion*, *reuse*, and *verification* of contents. These behaviors are controlled by headers.

2) Intermediary: An intermediary is an entity on a communication path between a server and a client as described in Section II-C. Programs corresponding to the intermediary with a cache are a *proxy* and a *gateway* for HTTP/1.1. The proposed model includes both a proxy and a gateway. An intermediary does not generate a request or response by itself and just forwards the received requests and responses. Therefore, only when an intermediary utilizes a cache, the intermediary can reply to a request via the reusability of the cache without forwarding the request and receiving the response. Meanwhile, a proxy and a gateway can edit contents in their communications.

3) HTTP header: Headers included in the basic model are insufficient for expressing the behavior of a cache, and thus we introduce additional headers shown in Table I. The cache-control headers in Table I specify the behavior of a cache, and thus our model also enables the use of their options. We show the options available in the proposed model in Table II. The behaviors of each term in Table I and Table I conform to the specification of HTTP/1.1.

TABLE I
HEADERS IN THE PROPOSED MODEL

Header name	purpose
if-modified-since	Used in requests with conditions
if-none-match	Used in requests with conditions
etag	Value of contents in a response
last-modified	Last update date of contents in a response
age	Elapsed time of a response
cache-control	Control of cache behavior
date	Time of a response generation
expire	Expiration date of a response

TABLE II
OPTIONS OF CACHE-CONTROL HEADERS IN THE PROPOSED MODEL

Option name	purpose
max-age	Set expiration date of a response
smax-age	Set expiration date of shared cache (top priority)
no-cache	Reusable after verification
no-store	Storing a response is prohibited
no-transform	Editing contetns is prohibited
max-stale	Permissible duration after expiration
min-fresh	Remaining time until expiration
only-if-cached	Response by cache only
must-revalidate	Reusable after verification until expiration
proxy-revalidate	Same as must-revalidate (except for personal cache)
public	Able to store in shared cache
private	Able to store in personal cache

4) Browser: The basic model contains the restriction that only a write is available in memory regions of a browser to simplify the expression. However, under this restriction, behaviors such as the deletion and the verification of responses stored in a cache cannot be executed. Consequently, the basic model cannot express the behavior of a cache fully. The proposed model overcomes the restriction described above by using the proposed syntax of temporal logic.

5) Threat Model: Our threat model is based on that of the basic model. Consider the three attackers described in Section III and the behavior of users as bases, and introduce the attack capability about a cache mechanism and an intermediary in an attacker. We describe the attackers as follows:

- A web attacker can own multiple intermediaries, can only forward contents, and cannot edit contents or interrupt the communication. However, it can eavesdrop on contents during unencrypted communication. Moreover, it can introduce caches into owned clients, servers, and intermediaries, and it operates according to specifications.
- A network attacker has all of the abilities of a web attacker. In addition, it can falsify contents of unencrypted communication and interrupt the communication via an intermediary.
- A gadget attacker has all of the abilities of a network attacker. In addition, it can falsify contents and interrupt communication for an intermediary.

6) Security Requirements: The proposed model has two security requirements that are identical to those of the basic model. In addition, our security invariants contain a specification of a cache and that of an intermediary.

B. Class of Cache

We show a class that expresses a cache in Code 7. We define the `Cache` class as an abstract class, the `PrivateCache` class to express a cache for each user, and the `PublicCache` class to express a shared cache, such as a proxy. Furthermore, we introduce restrictions into the `Cache` class described in Code 8 as follows:

- Each cache belongs to a terminal in a network.
- A personal cache belongs to a client.
- A shared cache belongs to a server or an intermediary.

Code 7
CACHE CLASS

```
abstract sig Cache{}
sig PrivateCache extends Cache{}
sig PublicCache extends Cache{}
```

Code 8
RESTRICTIONS FOR CACHE CLASS

```
fact noOrphanedCaches {
  all c:Cache |
    one e:NetworkEndpoint | c = e.
          cache
}

fact PublicAndPrivate{
  all pri:PrivateCache | pri in
        HTTPClient.cache
  all pub:PublicCache | (pub in
        HTTPServer.cache) or (pub in
        HTTPIntermediary.cache)
}
```

We also construct the class of components in the web, as shown in Code 9, to introduce a cache mechanism. We added the `Cache` class to the `HTTPConfirmist` class in the basic model, containing a client, a server, and an intermediary in the HTTP. Each device can then own at most one cache.

Code 9
COMPONENTS WITH CACHE UTILIZING HTTP IN THE WEB

```
abstract sig HTTPConformist extends
      NetworkEndpoint{
  cache : lone Cache
}
```

C. Class for Expressing States of Cache

We define the `CacheState` class that expresses states of a cache at each point in the temporal axis, as shown in Code 10. This `CacheState` class is extended from the `State` class and can use a predicate about the proposed temporal logic. The invariants defined in the `State` class are a finite set of the `Cache` class defined in Code 7, and the variants are that of responses to express stored responses. We then define `CacheEqItem` to express invariants and `CacheDifItem` to express variants in Code 10.

Moreover, we give the following restrictions to `CacheState` although we omit them in Code 10. This is a condition that must be held when a response is stored in a cache and follows the specification of HTTP/1.1.

- When a response is stored in a personal cache, either one of a max-age option of a cache-control header or a pair of a date header and an expire header is included.
- When a response is stored in a shared cache, either one of a max-age option of a cache-control header, an s-maxage option, or a pair of a date header and an expire header is included.
- When a response is stored in a cache, one age header is included.

Code 10
CLASS TO EXPRESS STATES OF CACHE

```
sig CacheState extends State{}{
  eq in CacheEqItem
  dif in CacheDifItem

  ...
}
sig CacheEqItem extends EqItem{cache:
      one Cache}
sig CacheDifItem extends DifItem{
     store: set HTTPResponse}
```

Next, we introduce the following restrictions that do not affect analysis results directly but can possibly reduce the number of instances to make the analysis easy.

- There are no multiple instances of a `State` class whose contents are identical to each other. When multiple instances have the same states at different times, the states are unified in the same `State` class.
- When a device with a cache communicates, a state of the cache is always expressed as an instance.
- When a response is stored in a cache, one age header is always included.

D. Behaviors of a Cache

Using the `CacheState` class defined in Section V-B, we express three basic behaviors of a cache described in Section V-A in Alloy.

1) Storage and Deletion of Responses: In the proposed model, storage and deletion of responses in a cache are executed as follows:

Definition 3 (Storage of Responses): A cache can store responses that its device sends and receives. The time to store in the cache is the same as that to send and receive the responses. Furthermore, let the stored responses meet all the conditions to be stored.

Definition 4 (Deletion of Responses): A cache can delete stored responses from the cache at any time.

We express the storage and deletion of responses as state transitions between two states, as shown in Fig. 5. The storage of responses for any state of each cache can be represented by adding the responses to the stored responses-so-far. This process corresponds to state transitions from CacheState0 to CacheState1 in Fig. 5. In this figure, CacheState0 has CacheDifItem0 and CacheState1 has CacheDifItem1 as variants, respectively. A set of the stored responses for CacheDifItem0 is an empty set while that for CacheDifItem1 includes Response0, indicating that a response for StateTransaction0 is Response0 and the response is stored in Cache0. Similarly, the deletion of responses can be expressed by not succeeding the responses from the previous state to the next state. This process corresponds to state transitions from CacheState1 to CacheState2 in Fig. 5. In this figure, CacheState2 includes CacheDifItem0 as a variant, and hence is identical to a reverse process of the storage of responses, indicating that Response0 stored in Cache0 at CacheState1 is deleted at CacheState2.

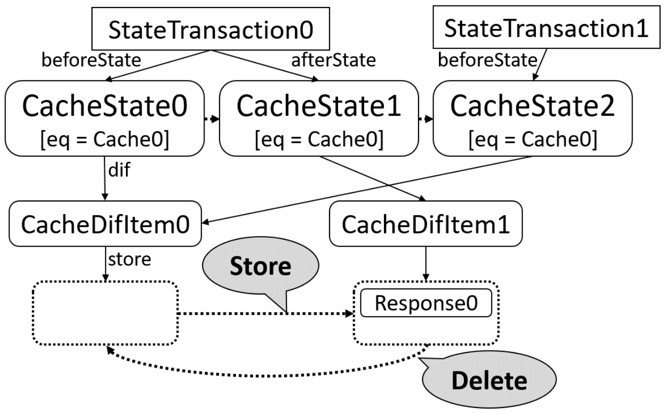

Fig. 5. Representation of Storage and Deletion in a Cache

The storage and deletion in a cache implemented in the proposed model are shown in Code 11. For any `post` of all `CacheState` classes, the previous state `pre` is obtained using a predicate `LastState`. When `post` is a state for a request, a set of the stored responses at the `post` is identical to a complement set for the set of the stored responses at the `pre`. Moreover, we represent the deletion of the stored responses where a part of the responses in the previous state is removed, i.e., its complement set. If `post` is a set at the response, a set of the stored responses at the `post` becomes a complement set where the response at the `post` is added to a set of the stored responses at the `pre`. Namely, adding a response at the `pre` to a set means storing of the response in a cache. However, we note that the states of a cache at each time depend on their previous states and hence the initial states always become unconditional. In this case, when the initial states are identical to states such that any communication has never been executed, there is no response in a cache at the initial states. Therefore, we express a restriction to assume a set of the stored responses at the initial state as an empty set by using `InitialState`.

Code 11
EXPRESSION OF STORAGE AND DELETION OF A CACHE

```
fact flowCacheState{
  all cs:CacheState |
    InitialState[cs] implies
```

```
      no cs.dif.store

  all pre, post:CacheState, str:
        StateTransaction |
    LastState[pre, post, str] implies
          {
      post in str.beforeState implies
             post.dif.store in pre.dif
           .store
      post in str.afterState implies
           post.dif.store in (pre.dif
           .store + str.response)
    }
}
```

2) Reuse of Responses: We describe the behavior of the reuse of responses by a cache in the proposed model as follows:

Definition 5 (Reuse of Responses): For any sent and received request, if a response for the request is stored in a cache, then a device with the cache can reply to the request by utilizing the response. A sender of the request reuses responses within its own cache, and the request is not sent to the network.

To do this, we define `CacheReuse` class in the proposed model similarly to the `HTTPResponse` class that represents responses in the basic model. The `CacheReuse` class is shown in the fourth line of Code 12. The `HTTPResponse` class is extended from the `HTTPEvent` class which expresses events in the HTTP protocol, and the `CacheReuse` class is extended from the `HTTPEvent` class. In addition, the `CacheResuse` class represents which response is reused by associating the class with a single response in the `HTTPResponse` class. Devices related to an event are defined in the `HTTPEvent` class, and thus a sender and a receiver of the reused responses can be expressed via the `HTTPEvent` class. We note that the `HTTPEvent` class includes a header and a body in addition to a sender and a receiver. For an actual reuse of responses, where a header and a body of the responses to be reused are sent, these two terms are typically unnecessary because they are expressed by associating them with the reuse of the responses described above. In other words, the relationship between a header and a body in the `CacheReuse` class can be ignored. Headers and bodies have many instances, and hence the computation in the proposed model spends a significant amount of time in proportion to the number of elements. To decrease computation time, we assume that both a header and a body in the `CacheReuse` class are empty sets, as shown in the seventh and eighth lines of Code 12, respectively.

Code 12
CACHEREUSE CLASS

```
1  sig HTTPResponse extends HTTPEvent {
2    statusCode: one Status
3  }
4  sig CacheReuse extends HTTPEvent{
5    target: one HTTPResponse
6  }{
7    no headers
8    no body
9  }
```

To express a reuse via the `CacheReuse` class described above, we construct the condition for the `CacheReuse` event along with an actual behavior of a cache as follows. First, a receiver of a reused response is the sender of the request that causes the reuse. A sender of a reused response is either the sender of the request that causes the reuse or a receiver of the request. Next, any previous state of the reuse of a cache includes a response to be reused in the stored responses. Finally, a URI for any request that causes the reuse is identical to that for any response to be reused.

We allow a sender of a reused response to be a sender of the request on the conditions described above because the sender may reuse responses in its own cache for any request.

3) Verification of Stored Responses: In the proposed model, verification of stored responses in a cache is defined as follows:

Definition 6 (Verification of Stored Responses): A device with a stored response sends a conditional request to an origin server to decide whether the response can be reused. A conditional request includes either an if-modified-since header or an if-none-match header. The origin server can decide whether the stored response is identical to the latest content by the use of a value transmitted with these headers and responds to the reuse of the response accordingly. If the verification finishes successfully, a status code of this response becomes 304 or 200. The status code of 304 indicates that the cache can reuse a stored response regardless of the values of a header and a body, and the status code 200 indicates that the stored response cannot be reused and the newly received response is stored in a cache. Moreover, in each case, except for the reusable responses, there is no response including the same URI in the cache after verification.

The behavior described above can be implemented by the following information:

- A predicate that decides whether a reused response has been verified.
- Behavior of a server for conditional requests.

The predicate `checkVerification`, which decides whether a response has been verified, is shown in Code 13. An input of the predicate is `StateTransaction`, which we refer to as `str`. The predicate is defined as follows: it is true if `str` is a transaction replied by the reuse, and if the transaction including a conditional request exists between the reuse and the request for `str`. This can be expressed as follows:

Definition 7 (Predicate of checkVerification): The predicate is true if an instance `str` in a `StateTransaction` class is reused and if there is an instance `str`' in a `StateTransaction` class that satisfies the following conditions:

- `str` and `str`' are different transactions;
- a response of `str`' exists, that is, the communication finished successfully;
- the request of `str`' occurs after `str`, and the response of `str`' occurs before the reuse of `str`;

- the request of `str`' is sent to the original sender for the reused response from a device whose cache stores the reuse of `str`;
- the requested URI for `str`' is identical to that for `str`;
- either an etag header or a last-modified header is included in the stored response to be verified;
- if an etag header is included in the stored response to be verified, an if-none-match header is included in a request for `str`'; and
- if a last-modified header is included in the stored response to be verified, and an if-modified-since header is included in a request for `str`'.

Code 13
PREDICATE THAT DECIDES WHETHER THE REUSE WAS VERIFIED

```
pred checkVerification[str:
      StateTransaction]{
  one str.re_res

  some str':StateTransaction |
  {
    str' != str
    one str'.response

    str'.request.current in str.
         request.current.*next
    str.re_res.current in str'.
         response.current.*next

    str'.request.from = str.re_res.
         from
    str'.request.to = str.re_res.
         target.from
    str'.request.uri = str.request.
         uri

    some h:HTTPHeader |{
      h in ETagHeader +
           LastModifiedHeader
      h in str.re_res.target.headers
    }

    (some h:ETagHeader | h in str.
         re_res.headers) implies
      (some h:IfNoneMatchHeader | h
           in str'.request.headers)
    (some h:LastModifiedHeader | h in
         str.re_res.headers) implies
      (some h:IfModifiedSinceHeader |
           h in str'.request.headers
           )
  }
}
```

Next, we express behavior of a server for any conditional request in Alloy. When an instance `tr` of the `HTTPTransaction` class is a communication that includes a conditional request, the server receiving the request adds the following conditions:

- after verification, there is exactly a single stored response that has a URI of the response to be verified;
- a status code of a response for `tr` is 200 or 304;
- if the status code is 200, the response for `tr` is stored in a cache; and
- if the status code is 304, the response for `tr` is not stored in a cache.

E. Implementation of Intermediary

In the proposed model, an intermediary works in both HTTP and HTTPS and includes a proxy and a gateway. We define a class of the intermediary `HTTPIntermediary` that extends `HTTPConfirmist`, whose device is according to HTTP. Likewise, we define a class of a proxy and a gateway that extends `HTTPIntermediary`. These classes are implemented as follows:

Code 14
CLASS OF INTERMEDIARY

```
abstract sig HTTPIntermediary extends
         HTTPConformist{}
sig HTTPProxy extends
       HTTPIntermediary{}
sig HTTPGateway extends
       HTTPIntermediary{}
```

Next, we express behavior of an intermediary in Alloy as follows: a receiver of a request is `HTTPIntermediary`, and the intermediary has `HTTPTransaction` whose response exists. For any instance `tr` of such `HTTPTransaction`, there is at least one instance `tr`' of `HTTPTransaction` that satisfies the following conditions:

- `tr` and `tr`' are different transactions;
- the request for `tr`' occurs after the request for `tr`, and a response for `tr`' occurs before a response for `tr`;
- a sender of the request for `tr`' is an intermediary that is the sender of the request for `tr`;
- a URI of the request for `tr`' is identical to that for `tr`; and
- a status code of a body of a response for `tr`' is identical to that for `tr`.

The behavior described above is that of an intermediary managed by a legitimate user, and the behavior of an intermediary managed by an attacker may be different.

VI. CASE STUDIES

In this section, we show several case studies related to analysis of web security. We discuss two basic behaviors of a cache mechanism and four attacks. The two behaviors will be used to confirm the capability of the proposed model. The four attacks, namely, verifications of a same-origin browser cache poisoning (BCP) attack [2], a cross-site request forgery (CSRF) attack [1], a cross-origin BCP attack [2], and a web cache deception attack [3], will be discussed to show improvements in the expressiveness of the proposed model. Hereinafter, for output of each case study, we summarize the original output of the Alloy Analyzer to help readers understand easily given that reading the original output of the Alloy Analyzer can be difficult due to a complicated state transition diagram. The complete original output can be obtained by executing our published code (**https://github.com/sho-rong/webmodel**).

A. Basic Behaviors of a Cache

In this section, we confirm the feasibility for behaviors of a cache in the proposed model. A cache executes three behaviors, i.e., storage, reuse, and verification, and we check each result using the Alloy Analyzer.

1) Storage of Responses: To check the behavior of storage of responses, we extract a result of storing of responses from an execution result using the Alloy Analyzer. For convenience, we obtained the result using Code 15 by targeting the storage of responses in the simplest communication, i.e., communication between two nodes. The results show that, for any communication with a pair of a request and a response between a client and a server, there is a state of a cache whose element is in a set of the stored responses.

Code 15
STORAGE OF RESPONSES

```
run test_store{
  #HTTPClient = 1
  #HTTPServer = 1
  #HTTPRequest = 1
  #HTTPResponse = 1
  some CacheState.dif.store
} for 2
```

Fig. 6 shows a simplified figure of the obtained results. This figure represents a state transition of a cache via a transaction between Request0 and Response0 for Browser0 with PrivateCache0 and Server0. Request0 is a transaction from Browser0 to Server0 while Response0 is the opposite. A state of a cache at Request0 is shown as CacheState0. Then, the state is changed to CacheState1 at Response0, and the corresponding CacheDifItem1 shows the store action for Response0. Fig. 6 represents a state where a response is stored in a browser cache for any transaction, and hence we can confirm that the storage of responses is expressible in the proposed model.

2) Reuse of Stored Responses: To check the behavior of the reuse of responses, we extract a result of storing of responses from an execution result using the Alloy Analyzer. For convenience, we obtained the result using Code 16 by targeting the reuse of responses in the simplest communication, i.e., communication between two nodes. The results show that, for any communication with a pair of a request and a response between a client and a server, one reuse of a single response occurs.

Code 16
REUSE OF STORED RESPONSES

```
run test_reuse{
  #HTTPClient = 1
  #HTTPServer = 1
  #Cache = 1

  #HTTPRequest = 2
  #HTTPResponse = 1
  #CacheReuse = 1
} for 4
```

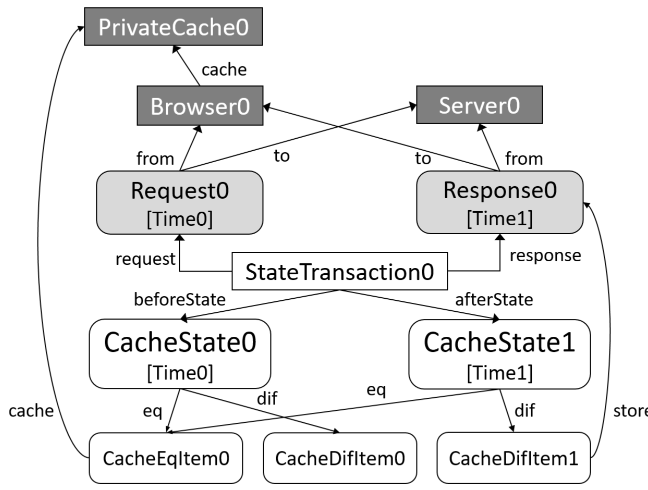

Fig. 6. Example of State for Storage of Responses

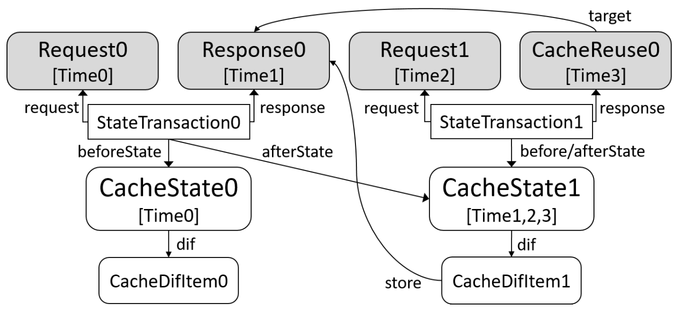

Fig. 7. Example of Reuse of Stored Responses

Fig. 7 shows a simplified figure of the original output. Fig. 7 represents a situation where two requests, Request0 and Request1, are sent to the same URI between a browser with a cache and a server. In addition, StateTransaction0 is storing Response0 in a browser cache in a similar manner as in Fig. 6. Then, for StateTransaction1, CacheReuse0 which is an event for the reuse of the stored response calls Response0 with the target. These results confirm that the behavior of the reuse is expressible in the proposed model.

3) Verification of Stored Responses: To check verification of responses, we extract the results from an execution result using the Alloy Analyzer. For convenience, we obtained the results using Code 17 by targeting the verification of responses in the simplest communication, i.e., communication between two nodes. The results show communication with verification for any communication with a pair of a request and a response between a client and a server. We also decide whether verification is performed or not via the predicate described in Section V-D3, i.e., Code 13.

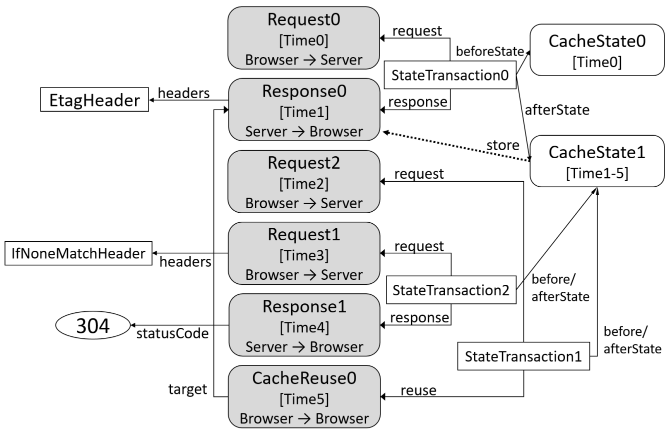

Fig. 8. Example of Verification for Stored Responses

Code 17
VERIFICATION OF STORED RESPONSES

```
run test_verification{
  #HTTPClient = 1
  #HTTPServer = 1
  #HTTPIntermediary = 0
  #Cache = 1
  #PrivateCache = 1

  some str:StateTransaction |
       checkVerification[str]
} for 6
```

Fig. 8 shows a simplified figure of the original output. Fig. 8 represents a situation where three communications, i.e., StateTransaction0, StateTransaction1, and StateTransaction2, occur between a browser with a cache and a server, and only StateTransaction2 is the communication with verification. First, the browser stores Response0 in its cache by StateTransaction0 similarly to Fig. 6. Next, the browser verifies in StateTransaction2 to reuse Response0 stored in the cache for Request2. Request1, which is a conditional request for the verification, includes an IfNoneMatchHeader because the stored response, i.e., Response0, includes an EtagHeader. Then, the verification result for Request1, i.e., Response1, contains a 304 status code, which means the reuse is available, and hence Response0 is reused. The process described above therefore shows that Response0 was reused in StateTransaction1 and the verification finished successfully. These results confirm that behavior of the verification is expressible in the proposed model.

B. Basic Behaviors of an Intermediary

To check behavior of intermediaries, we extract a result for verification of responses from an execution result using the Alloy Analyzer. For convenience, we obtained the result using Code 18 by targeting communication transited via an intermediary in the simplest case with a single client, a single proxy, and a single server.

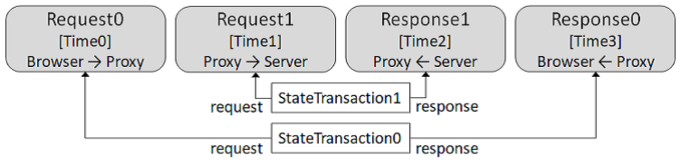
Request0 [Time0] Browser → Proxy
Request1 [Time1] Proxy → Server
Response1 [Time2] Proxy ← Server
Response0 [Time3] Browser ← Proxy
request
StateTransaction1
response
request
StateTransaction0
response

Fig. 9. Example of Behavior of Intermediaries

Code 18
BEHAVIOR FOR INTERMEDIARY

```
run test_intermediary{
  #HTTPRequest = 2
  #HTTPResponse = 2

  #HTTPClient = 1
  #HTTPServer = 1
  #HTTPIntermediary = 1

  all i:HTTPIntermediary | i in Alice
       .servers

  one req:HTTPRequest | req.to in
       HTTPIntermediary
  one req:HTTPRequest | req.to in
       HTTPServer
} for 4
```

Fig. 9 shows a simplified figure of the original output. Fig. 9 represents a communication between a browser of the client and the proxy as StateTransaction0 and that between the proxy and the server as StateTransaction1. The intermediary, i.e., the proxy, forwards received requests and responses to these destinations and, in particular, StateTransaction1 is the forwarding information for StateTransaction0. These results confirm that the behavior of an intermediary is expressible in the proposed model.

C. Same-origin Browser Cache Poisoning Attack

In this section, we show that our proposed model can express a same-origin browser cache poisoning (BCP) attack. Same-origin BCP attack is a man-in-the-middle attack where an attacker interposes an intermediary between a target browser and a server. The main purpose of this attack is to store information generated by the attacker in the browser as a response from the server, and to execute arbitrary behavior designated by the attacker against the browser. The main advantage of this attack is continuousness, i.e., even by a single interruption to communication, the target browser is affected whenever the response is reused.

The flow of same-origin BCP attack is shown in Fig. 10. In this attack, the attacker manipulates a response generated for a browser and then stores the manipulated response in a target browser. The attacker manipulates two data, i.e., a header and a body. For a header of a response, the attacker manipulates in a way that the response is stored and reused by a browser cache. For instance, an attacker may extend an expiry date or

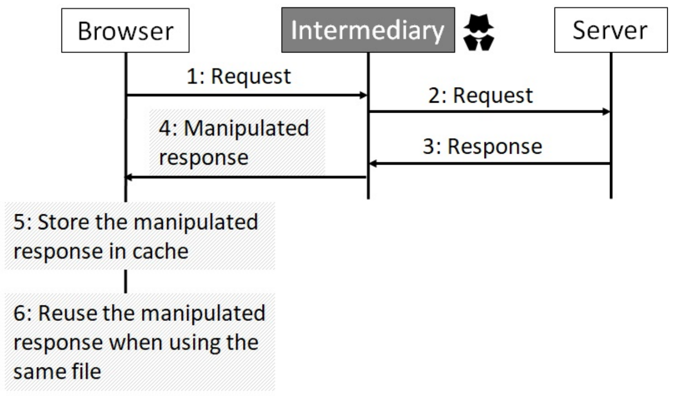
Browser
Intermediary
Server
1: Request
2: Request
3: Response
4: Manipulated response
5: Store the manipulated response in cache
6: Reuse the manipulated response when using the same file

Fig. 10. Flow of Same-origin Browser Cache Poisoning Attack

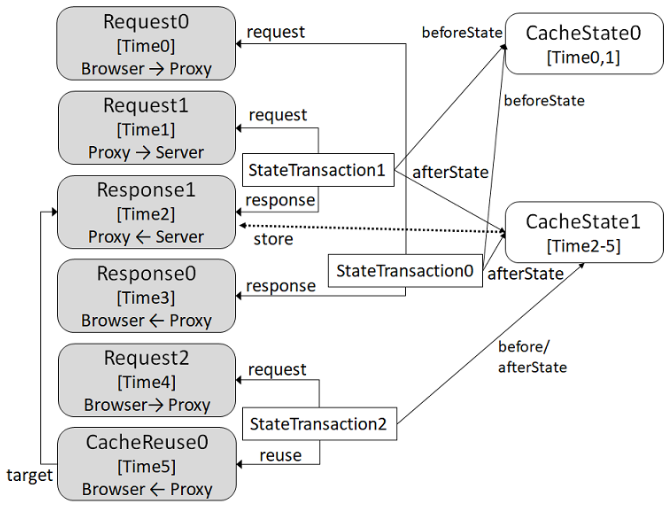
Request0 [Time0] Browser → Proxy
Request1 [Time1] Proxy → Server
Response1 [Time2] Proxy ← Server
Response0 [Time3] Browser ← Proxy
Request2 [Time4] Browser→ Proxy
CacheReuse0 [Time5] Browser ← Proxy
CacheState0 [Time0,1]
CacheState1 [Time2-5]
StateTransaction1
StateTransaction0
StateTransaction2
request
response
store
reuse
beforeState
afterState
before/ afterState
target

Fig. 11. Example of Same-origin Browser Cache Poisoning Attack

reuse a response without verification. For a body of a response, the attacker describes arbitrary behavior to be executed for a victim.

In Fig. 10, the browser is affected when a manipulated response is stored in a cache and when the manipulated responses is reused to access the same file, i.e., the fifth and sixth phases. The sixth phase may be iterated as long as the manipulated response is stored in the cache.

The same-origin BCP attack is expressed in the code shown in Appendix A. Fig. 11 shows a simplified figure of the original output. Fig. 11 represents a communication between a browser of a client and a proxy as StateTransaction0 and that between the proxy and a server as StateTransaction1. An intermediary, i.e., the proxy, forwards received requests and responses to these destinations, and StateTransaction0 is the forwarding communication for StateTransaction0. However, the proxy is a device owned by the attacker, and the response forwarded via the proxy manipulates the original response generated from the server. In particular, the browser stores the manipulated response stored in CacheState0 and then reuses the response in StateTransaction2. We can thus confirm that a same-origin BCP attack is expressible in the proposed model.

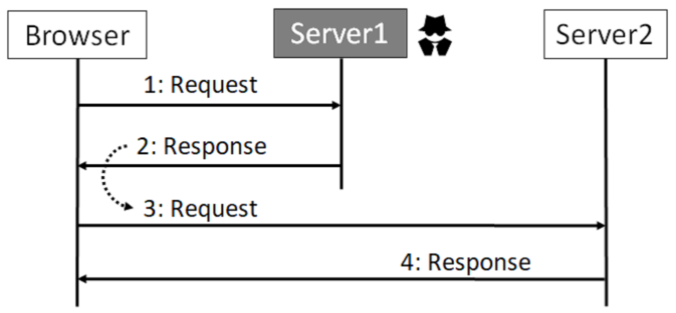

Fig. 12. Flow of Cross-site Request Forgery Attack

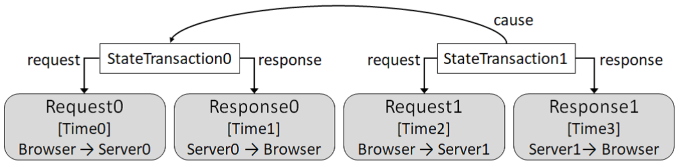
cause
request
StateTransaction0
response
request
StateTransaction1
response
Request0 [Time0] Browser → Server0
Response0 [Time1] Server0 → Browser
Request1 [Time2] Browser → Server1
Response1 [Time3] Server1→ Browser

Fig. 13. Example of Cross-Site Request Forgery Attack

D. Cross-site Request Forgery Attack

In this section, we show that our proposed model can express a cross-site request forgery (CSRF) attack [5]. The CSRF attack is executed among three parties, i.e., a target server, a server managed by an attacker, and a browser of a client. The main purpose of this attack is to execute arbitrary behavior for the target server.

The flow of the CSRF attack is shown in Fig. 12. Server1 is a server managed by an attacker and Server2 is a target server, i.e., a victim. The attacker behaves as follows: when Server1 receives a request from a browser, the server returns a response such that the browser is forced to send another request, i.e., the request on the third phase in Fig. 12. Furthermore, the request sent from the browser is also operated along with the response returned from Server1.

The CSRF attack is expressed in the code shown in Apendix B. Fig. 13 shows a simplified figure of the original output. Server0 is a server managed by an attacker and Server1 is a target server, i.e., a victim. StateTransaction0 is a communication between a browser and Server0, while State-Transaction1 is a communication between the browser and Server1. The figure shows that StateTransaction1 is a relation with the cause for StateTransaction0, i.e., StateTransaction1 is caused by StateTransaction0. These results confirm that a CSRF attack is expressible in the proposed model.

E. Cross-origin Browser Cache Poisoning Attack

In this section, we show that our proposed model can express a cross-origin browser cache poisoning (BCP) attack. In this kind of BCP attack, an attacker manipulates a response communicated between multiple servers, e.g., redirection between different servers. In other words, the attacker can also focus on files which are utilized in multiple sites, e.g., a css file or a js file.

The flow of the cross-origin BSP attack is shown in Fig. 14. Server1 and Server2 are honest servers, while Intermediary is

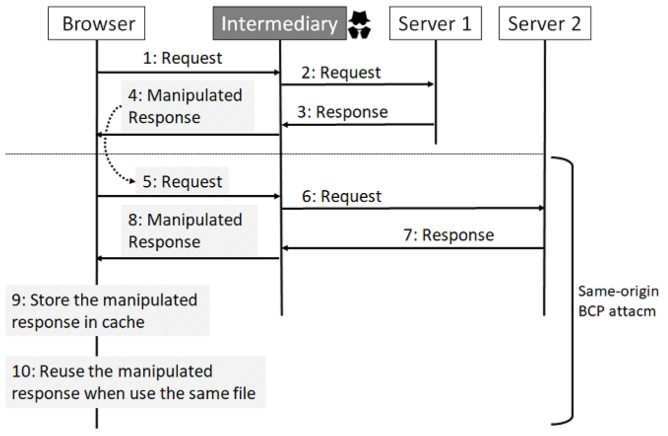

Fig. 14. Flow of Cross-origin Browser Cache Poisoning Attack

a device managed by an attacker. The phases after the fifth phase are identical to those of the same-origin BCP attack. Likewise, phases before the fifth phase are similar to those of the CSRF attack. In particular, the attacker first manipulates a response by interrupting communication during the first phase. The manipulation aims to cause a request on the fifth phase for a target file. For example, when there is a file named A.css used in many pages, a request for A.css can be caused by describing the use of A.css at a response on the third phase. The attacker can then manipulate and store an arbitrary file in a browser cache. Moreover, even if Server1 and Server2 have new communication after a successful attack, a browser will be affected when a file designated in the manipulated response is reused.

The attack is expressed in the code shown in Appendix C. As a result of execution, our proposed model can output a situation including the five communications shown in Fig. 14. Fig. 15 shows a simplified figure of the original output. In Fig. 15, the process is identical to that of Fig. 11 until Time3. At Time4 and Time5, the process is redirected to another server, and an attacker manipulates Response3 from Server2 at Time6. A cache stores the manipulated response in a cache, and then a cache state transitions from CacheState0 to CacheState1. The stored response is reused in StateTransaction4. The output represents a situation where a manipulated response is reused after a redirection. These results confirm that a cross-site request forgery attack is expressible in the proposed model.

F. Web Cache Deception Attack

In this section, we show that our proposed model can express a web cache deception (WCD) attack [3]. The WCD attack is executed among a target server, a target intermediary, and two browsers. An attacker owns one of the browsers and tries to extract a user's file that is unavailable to the attacker.

Fig. 16 shows the flow of the WCD attack. Browser1 is a browser managed by an attacker and Browser2 is a victim browser. The victim browser accesses a file that is unavailable to the attacker via the intermediary. When the intermediary stores a file returned as the response in a cache on the third phase, the attacker then sends a request for the file to the intermediary and can extract the file via the reuse of the file.

The attack is expressed in the code shown in Appendix D. Fig. 17 shows a simplified figure of the original output. The figure is similar to Fig. 11. In Fig. 17, a request on the first phase and that on the second phase in Fig. 16 correspond to StateTransaction0 and StateTransaction1, respectively. Token, which is unavailable to the attacker, is attached to a body of Response1. Moreover, Response1 is stored in a cache of the intermediary at Time2 and reused for StateTransaction2, which is communication by the attacker. The output therefore represents a situation where an attacker can extract a file that is unavailable for the attacker itself by the reuse via a cache of an intermediary. These results confirm that a WCD attack is expressible in the proposed model.

VII. Conclusion

In this paper, we aimed to use formal methods to perform security analysis of the web. Therefore, we proposed a new web security model including a cache that can be used for security analysis of web services, such as social network services (SNS). We presented a new syntax of temporal logics in Alloy to express state transitions of elements in the web.

In the proposed syntax, we defined a versatile class that can express various states of elements in the web. Furthermore, we implemented two predicates that handle instances for a class along with time series. We can hence express state transitions between state classes in different communications, which are inexpressible in current works.

Then, we verified that our proposed model can express basic behavior of a cache mechanism. In addition, we verified that four attacks including state transitions that are inexpressible in current models, and hence confirmed improvements in the expressiveness of the proposed model. We consider that these attacks are important problems in web systems, such as SNS, and verification of these attacks will contribute in providing security for SNS. Although our source code does not include HTTPS, our model can be used for verification of HTTPS by extending it along with the specifications of HTTPS. We also consider that various vulnerabilities for not only a cache but also for other mechanisms can be verified by extending our model.

We will discuss countermeasures for attacks shown in the case studies as future work. In particular, we will discuss the countermeasures by finding conditions where the attacks fail in the proposed model. Moreover, we plan to apply our model to the analysis of the HTTP strict transport security (HSTS) [24] and the public key pinning extension for HTTP (HPKP) [25], which are extended protocols of HTTPS. These protocols are state-of-the-art protocols and their security analysis via formal methods is necessary. We consider that the security of these protocols can be analyzed by introducing their headers in our model.

Acknowledgments

This work was supported by JST ACT-I Grant Number JPMJPR16UQ and JSPS KAKENHI Grant Number 18K18049, Japan.

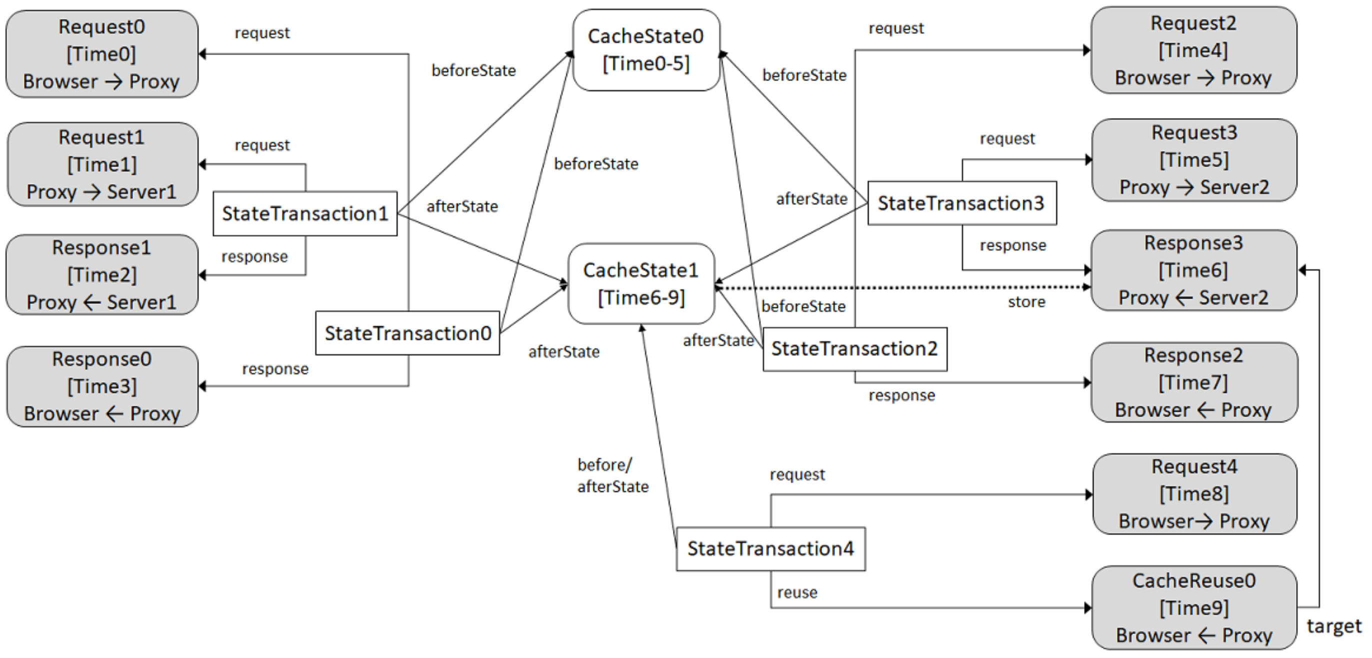
Request0 [Time0] Browser → Proxy
Request1 [Time1] Proxy → Server1
Response1 [Time2] Proxy ← Server1
Response0 [Time3] Browser ← Proxy
CacheState0 [Time0-5]
CacheState1 [Time6-9]
StateTransaction0
StateTransaction1
StateTransaction2
StateTransaction3
StateTransaction4
Request2 [Time4] Browser → Proxy
Request3 [Time5] Proxy → Server2
Response3 [Time6] Proxy ← Server2
Response2 [Time7] Browser ← Proxy
Request4 [Time8] Browser→ Proxy
CacheReuse0 [Time9] Browser ← Proxy
request
response
beforeState
afterState
store
before/ afterState
reuse
target

Fig. 15. Example of Cross-Origin Browser Cache Poisoning Attack

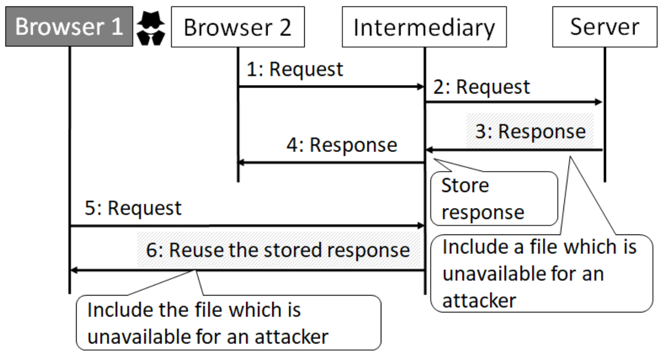
Browser 1
Browser 2
Intermediary
Server
1: Request
2: Request
3: Response
4: Response
Store response
5: Request
6: Reuse the stored response
Include a file which is unavailable for an attacker
Include the file which is unavailable for an attacker

Fig. 16. Flow of Web Cache Deception Attack

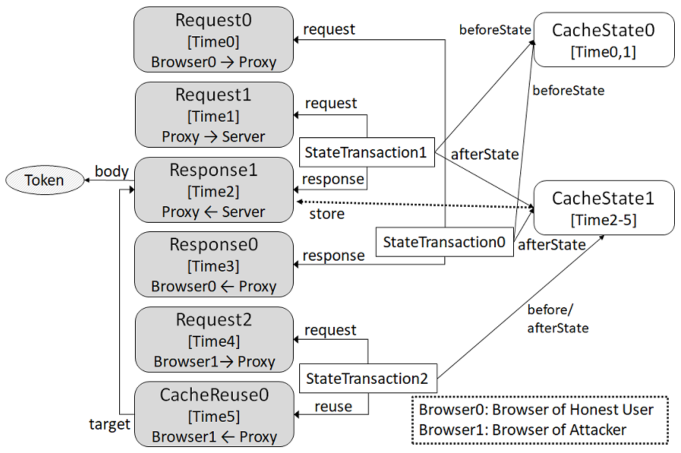
Request0 [Time0] Browser0 → Proxy
Request1 [Time1] Proxy → Server
Response1 [Time2] Proxy ← Server
Response0 [Time3] Browser0 ← Proxy
Request2 [Time4] Browser1→ Proxy
CacheReuse0 [Time5] Browser1 ← Proxy
CacheState0 [Time0,1]
CacheState1 [Time2-5]
StateTransaction0
StateTransaction1
StateTransaction2
Token
body
request
response
beforeState
afterState
store
before/ afterState
reuse
target
Browser0: Browser of Honest User
Browser1: Browser of Attacker

Fig. 17. Example of Web Cache Deception Attack

REFERENCES

[1] OWASP. Cross-site request forgery (csrf), 2018.

[2] Yaoqi Jia, Chen Yue, Dong Xinshu, Saxena Prateek, Mao Jian, and Liang Zhenkai. Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning. In *Computers and Security*, Vol. 55, pp. 62–80, 2015.

[3] Reo Ogawa, Yuya Okuda, and Taiichi Saito. Web Cache Deception Vulnerability Scanner. In *Symposium on Cryptgraphy and Information Security*, pp. 23–26, 2018. (in Japanese).

[4] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *IEEE Computer Security Foundations Symposium*, pp. 290–304, 2010.

[5] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and Precise Client-Side Protection against CSRF Attacks. In *European conference on Research in computer security*, pp. 100–116, 2011.

[6] Krishna Chaitanya, Akash Agrawall, and Venkatesh Choppella. A Formal Model of Web Security Showing Malicious Cross Origin Requests and Its Mitigation using CORP. *International Conference on Information Systems Security and Privacy*, pp. 516–523, 2017.

[7] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Dependable Systems and Networks*, pp. 514–525, 2016.

[8] Kevin Zhijie Chen, Warren He, Devdatta Akhawe, Vijay D'Silva, Prateek Mittal, and Dawn Song. ASPIRE: Iterative Specification Synthesis for Security. In *HotOS*, 2015.

[9] Tim Nelson, Salman Saghafi, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: principled scenario exploration through minimality. In *International Conference on Software Engineering*, pp. 232–241, 2013.

[10] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *3rd ACM workshop on Cloud computing security workshop*, pp. 3–14, 2011.

[11] Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. Formal methods for web security. *Journal of Logical and Algebraic Methods in Programming*, Vol. 87, pp. 110–126, 2017.

[12] Eric Y. Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: Get the security of multiple browser with just one. In *ACM Conference on Computer and Communication Security*, pp. 227–237, 2011.

[13] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *Computer Security Foundations Symposium*, pp. 247–262, 2012.

[14] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and

Sergio Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *Principles of Security and Trust*, pp. 126–146, 2013.
[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browserid sso system. In *IEEE Symposium Security and Privacy*, pp. 673–688, 2014.
[16] Daniel Fett, Ralf Küsters, and Guido Schmitz. Analyzing the browserid sso system with primary identity providers using an expressive model of the web. In *European Symposium on Research in Computer Security*, pp. 43–65, 2015.
[17] Daniel Fett, Ralf Küsters, and Guido Schmitz. Acm conference on computer and communications security. In *A Comprehensive Formal Security Analysis of OAuth 2.0*, pp. 1204–1215, 2016.
[18] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *Computer Security Foundations Symposium*, pp. 189–202, 2017.
[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*, pp. 207–220, 2009.
[20] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. Towards Formal Analysis of the Permission-Based Security Model for Android. In *International Conference on Wireless and Mobile Communications*, pp. 87–92, 2009.
[21] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Security and Privacy*, pp. 166–177, 2003.
[22] Joseph P. Near and Daniel Jackson. Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns. In *IEEE International Conference on Software Engineering*, pp. 947–958, 2016.
[23] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. In *Security Advisory*, 2014.
[24] Jeff Hodges, Collin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS). RFC 6797, 2012.
[25] Chris Evans, Chris Palmer, and Ryan Sleevi. Public Key Pinning Extension for HTTP. RFC 7469, 2015.
[26] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Network and Distributed System Security Symposium*, 2015.
[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.
[28] Ali Ayad and Claude Marché. Multi-Prover Verification of Floating-Point Programs. In *5th International Joint Conference Automated Reasoning,*, pp. 127–141, 2010.
[29] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An Automated Prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research*, 2017.
[30] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. Towards Formal Verification of Contiki: Analysis of the AESCCM* Modules with Frama-C. In *Workshop on Recent advances in secure management of data and resources in the IoT*, 2018.
[31] Ivan Bocić and Tevfik Bultan. Symbolic model extraction for web application verification. In *International Conference on Software Engineering*, pp. 724–734, 2017.
[32] Jason Bau and John C. Mitchell. Security Modeling and Analysis. In *IEEE Symposium on Security and Privacy*, Vol. 9, pp. 18–25, 2011.
[33] Roy Fielding and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), 2014.
[34] Roy Fielding and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard), 2014.
[35] Roy Fielding and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232 (Proposed Standard), 2014.
[36] Roy Fielding, Yves Lafon, and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233 (Proposed Standard), 2014.
[37] Roy Fielding, Mark Nottingham, and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234 (Proposed Standard), 2014.
[38] Roy Fielding and Peter J Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235 (Proposed Standard), 2014.
[39] Tim Berners-Lee, Peter J Reschke, and Henrik Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), 1996.

APPENDIX A
VERIFICATION CODE FOR SAME-ORIGIN BROWSER CACHE POISONING ATTACK

Code 19
EXPRESSION OF SAME-ORIGIN BROWSER CACHE POISONING ATTACK

```
run Same_origin_BCP{
  #HTTPClient = 1
  #HTTPServer = 1
  #HTTPIntermediary = 1
  #PrivateCache = 1
  #PublicCache = 0

  #HTTPRequest = 3
  #HTTPResponse = 2
  #CacheReuse = 1

  #Principal = 3
  #Alice = 2

  some tr,tr',tr'':HTTPTransaction |
      {
    tr'.request.current in tr.request
        .current.*next
    tr.response.current in tr'.
        response.current.*next
    tr''.request.current in tr.
        response.current.*next
    some tr''.re_res

    tr.request.from in HTTPClient
    tr.request.to in HTTPIntermediary

    tr'.request.from in
        HTTPIntermediary
    tr'.request.to in HTTPServer

    tr''.request.from in HTTPClient

    tr.response.body != tr'.response.
        body
  }

  some c:HTTPClient | c in Alice.
      httpClients
  some s:HTTPServer | s in Alice.
      servers
  no i:HTTPIntermediary | i in Alice.
      servers
} for 6
```

APPENDIX B
VERIFICATION CODE FOR CROSS-SITE REQUEST FORGERY ATTACK

Code 20
EXPRESSION OF CROSS-SITE REQUEST FORGERY ATTACK

```
run CSRF{
  #HTTPRequest = 2
  #HTTPResponse = 2

  #HTTPClient = 1
  #HTTPServer = 2
  #HTTPIntermediary = 0

  #Principal = 3
  #Alice = 2

  all p:Principal |
    one c:HTTPConformist |
      c in p.(servers + httpClients)
  all b:Browser | b in Alice.httpClients

  one tr1,tr2:HTTPTransaction|{
    tr2.request.current in tr1.response.current.*next

    tr1.request.to !in Alice.servers
    tr2.request.to in Alice.servers

    tr2.cause = tr1

    tr1.request.uri != tr2.request.uri
  }
} for 4
```

APPENDIX C
VERIFICATION CODE FOR CROSS-ORIGIN BROWSER CACHE POISONING ATTACK

Code 21
EXPRESSION OF CROSS-ORIGIN BROWSER CACHE POISONING ATTACK

```
run Cross_origin_BCP{
  #HTTPRequest = 5
  #HTTPResponse = 4
  #CacheReuse = 1

  #Browser = 1
  #HTTPServer = 2
  #HTTPProxy = 1
  #Cache = 1

  #Principal = 4
  #Alice = 3

  one Uri

  all c:Cache | c in Browser.cache

  all p:Principal |
    one c:HTTPConformist |
      c in p.(servers + httpClients)
  all b:Browser | b in Alice.httpClients
  all s:HTTPServer | s in Alice.servers

  all tr:HTTPTransaction |{
    tr.request.to in HTTPIntermediary implies{
      one tr':HTTPTransaction |{
        tr'.request.from in HTTPIntermediary
        tr'.request.to in HTTPServer

        tr'.request.current = tr.request.current.next
        tr'.response.current = tr'.request.current.next
        tr.response.current = tr'.response.current.next
      }
    }
  }

  one disj tr1,tr3:HTTPTransaction |{
    tr1.request.from in HTTPClient
    tr1.request.to in HTTPIntermediary
    tr3.request.from in HTTPClient
    tr3.request.to in HTTPIntermediary

    tr3.request.current in tr1.response.current.*next
    tr3.cause = tr1
  }

  some tr2,tr4:HTTPTransaction |{
    tr2.request.from in HTTPProxy
    tr2.request.to in HTTPServer
    tr4.request.from in HTTPProxy
    tr4.request.to in HTTPServer
    tr2.request.to != tr4.request.to
  }

  one tr5:HTTPTransaction |{
    tr5.request.from in HTTPClient
    tr5.request.to in HTTPServer
    one tr5.re_res

    all tr:HTTPTransaction | (one tr.response implies tr5.request.current in tr.response.current.*next)
  }

  all tr,tr':HTTPTransaction |{
    {
      tr.request.from in HTTPClient
      tr.request.to in HTTPIntermediary
      tr'.request.from in HTTPIntermediary
      tr'.request.to in HTTPServer
    }implies{
      tr.response.body != tr'.response.body
    }
  }
} for 10
```

APPENDIX D
VERIFICATION CODE FOR WEB CACHE DECEPTION ATTACK

Code 22
EXPRESSION OF WEB CACHE DECEPTION ATTACK

```
run Web_Cache_Deception{
  #HTTPRequest = 3
  #HTTPResponse = 2
  #CacheReuse = 1

  #HTTPClient = 2
  #HTTPServer = 1
  #HTTPProxy = 1
  #Cache = 1

  #Principal = 4
  #Alice = 3

  all c:Cache | c in HTTPProxy.cache

  all p:Principal |
    one c:HTTPConformist |
      c in p.(servers + httpClients)
  all i:HTTPProxy | i in Alice.servers
  all s:HTTPServer | s in Alice.servers

  one tr1,tr2,tr3:HTTPTransaction |{
    tr1.request.from in Alice.httpClients
    tr1.request.to in HTTPProxy

    tr2.request.from in HTTPProxy
    tr2.request.to in HTTPServer

    (tr3.request.from !in Alice.httpClients and tr3.request.from in HTTPClient)
    tr3.request.to in HTTPProxy

    one tr3.re_res
  }
} for 6
```

PLACE
PHOTO
HERE

Shimamoto Hayato Biography text here.

PLACE
PHOTO
HERE

Naoto Yanai Biography text here.

PLACE
PHOTO
HERE

Shingo Okamura Biography text here.

PLACE
PHOTO
HERE

Jason Paul Cruz Biography text here.

PLACE
PHOTO
HERE

Shouei Ou Biography text here.