CAE Tech Talk

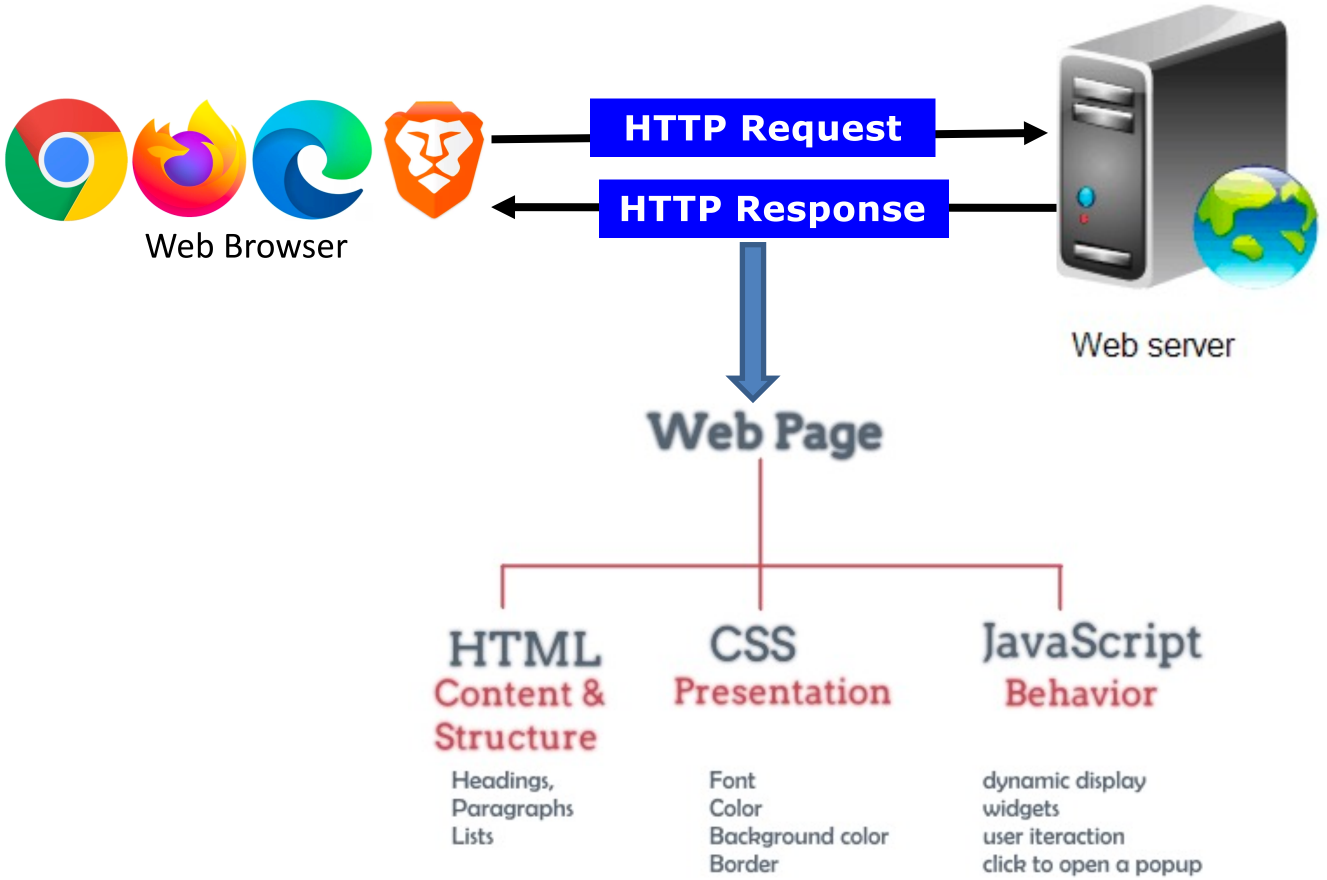
October 21, 2021

A User-Oriented Approach and Tool for Security and Privacy Protection on the Web

Phu H. Phung
Intelligent System Security Lab
https://isseclab-udayton.github.io
Department of Computer Science
University of Dayton

The foundation of the Web

- Based on the HTTP protocol
  - Regardless the Web technologies



JavaScript capabilities – in browsers

- Interact with users
- Modify webpages
- Read/write local data, e.g., cookies
- Send/receive data over the network

In-Browser JavaScript Security Review

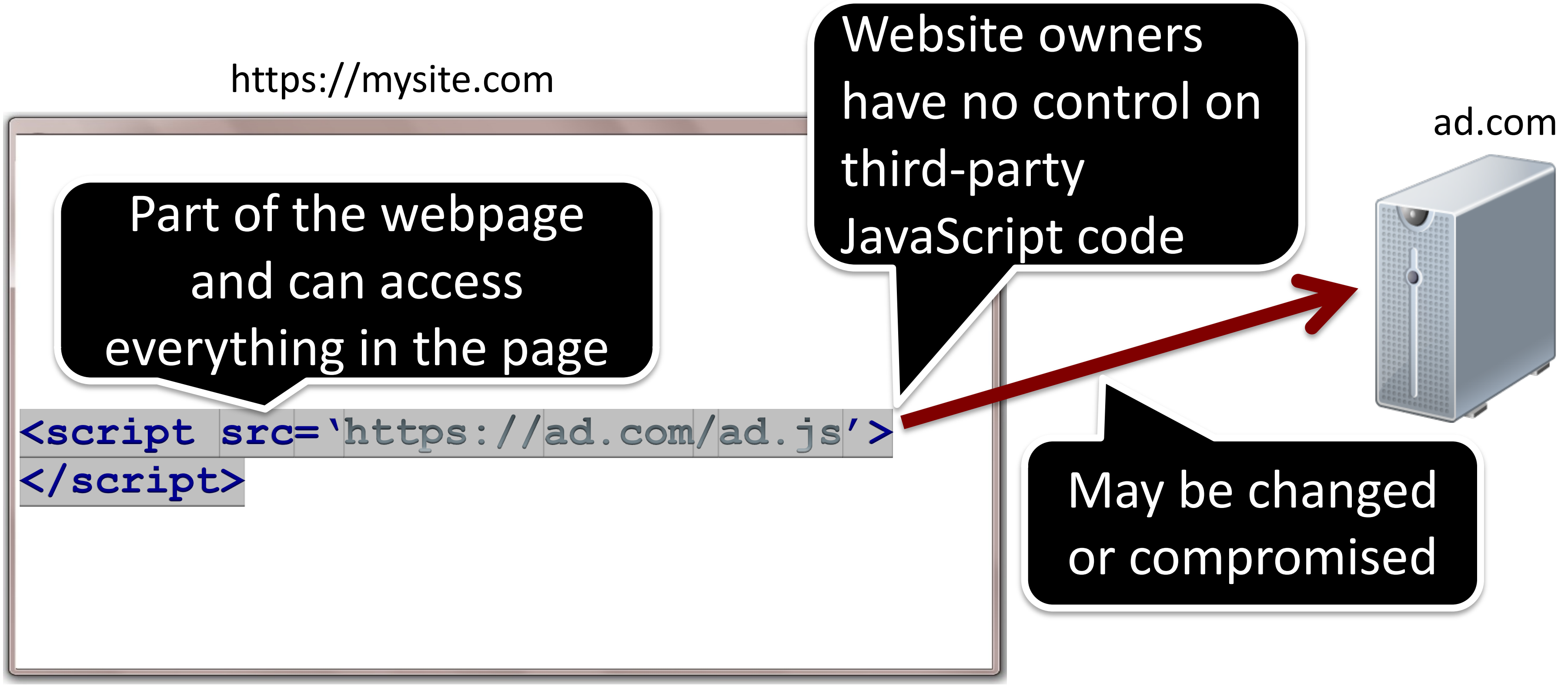
- JavaScript code is executed in Web Browsers (in a JavaScript Engine – Interpreter) under a “sandbox” environment
  - No direct file access, restricted network access

- JavaScript code is enforced by Same-Origin Policy (SOP)
  - Can only access (read/write) the properties of webpages from the same **domain, protocol, and port** (that form the origin)
    - E.g.: Code from https://ad.com CANNOT access data of https://mysite.com in the same browser

- Content Security Policy (CSP) is an additional layer of protection to prevent attacks such as Cross-Site Scripting (XSS) and data injection attacks

Limitations of SOP and CSP

- Still based on the trustworthy, i.e., should be whitelisted in CSP
  - Third-party code loaded from external source has the same origin policy as the hosting page



A Webpage example with third-party JavaScript

- Contains internal script code and includes external code
  - External/third-party code is normally trusted and included into webpages by the host/developer

"88.45% of the Alexa top 10,000 web sites included at least one external JavaScript code" [Nikiforakis et al, CCS'12]

Third-party JavaScript Problems

A Real Attack Example under SOP and CSP

- Attacks still happen with SOP and CSP security mechanisms. Example: A real attack on reuters.com

Reuters website was attacked by code injection via a compromised ad network.

Third-party JavaScript trusted and included by Reuters.com

Third-party JavaScript Security

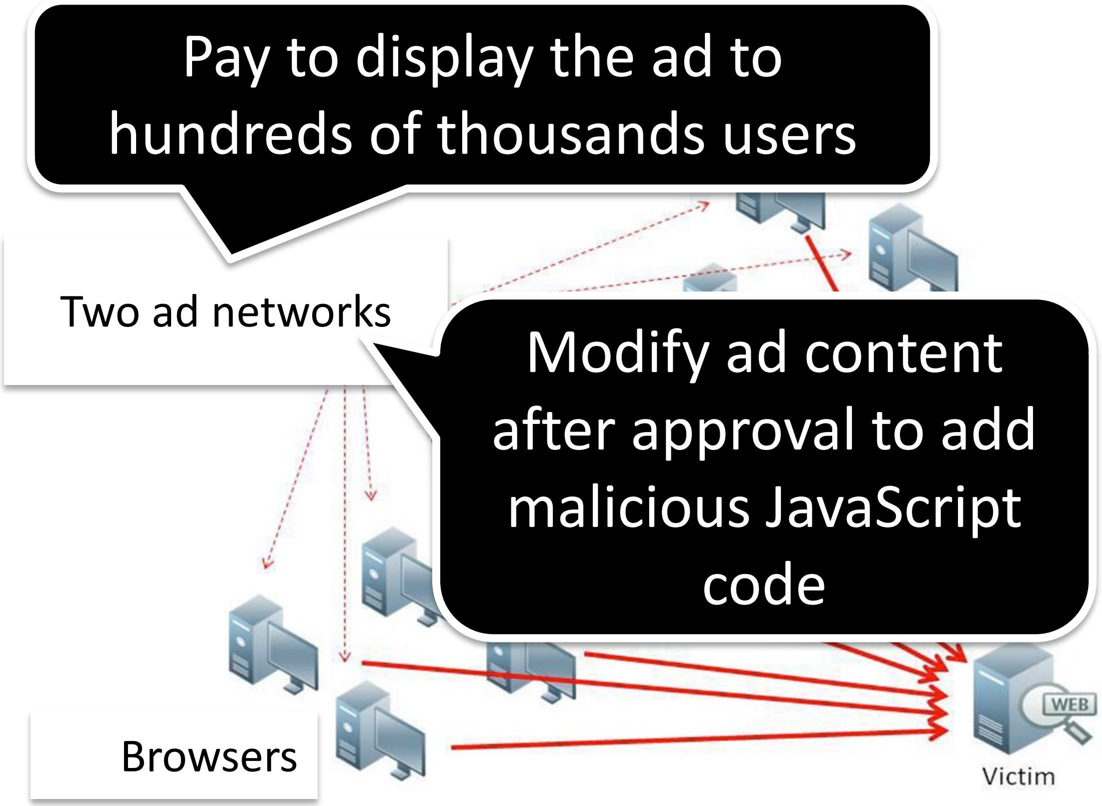
"The most reliable, cost effective method to inject evil code is to buy an ad"

-Douglas Crockford

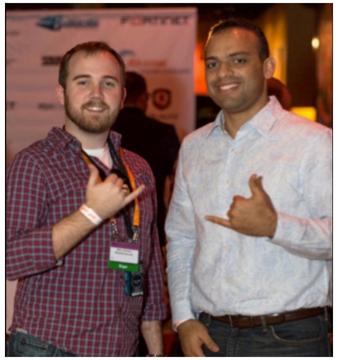
JavaScript Security Expert

A Research Attack

‘Million Browser Botnet’



DDoS attack
(Distributed Denial of Service)



WhiteHat SECURITY

The problem

- How to ensure that JavaScript code, either from first-party or third-party does not perform malicious actions on users’ devices?

Existing solutions and open challenges

- Short-term: *all-or-nothing approach*
  - Browser extension blockers
  - In-browser blockers
- Long-term: *no formal mechanisms to ensure the enforcement*
  - Do-Not-Track
  - Privacy by Design
  - W3C Platform for Privacy Preferences Project
  - Regulations
    - European Union’s General Data Protection Regulation (GDPR)
    - The U.S. State Privacy Laws
- More open challenges
  - Few prior work consider the issues of the same-origin policy, e.g., third-party code is malicious or compromised
  - Users has no or little control on their data from an end device
  - There is no formal assurance mechanism to guarantee that agreements/rules are enforced

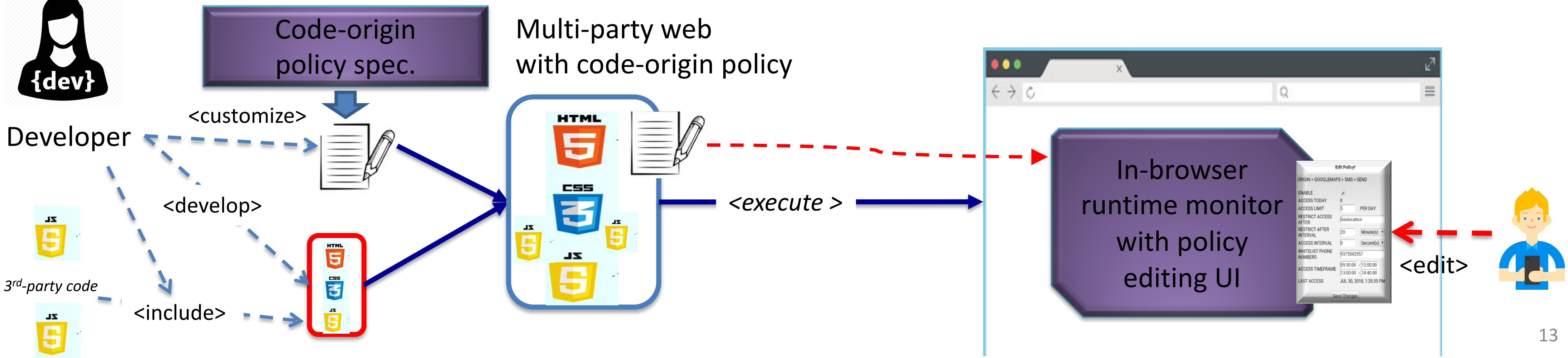
Concerns and Dilemma of Web Users

- Malicious/vulnerable websites exists and can compromise users' privacy and security, e.g., the Reuters.com example
- Citizens trust the big companies to not misuse their data [1,2]
- Several prior studies showed that portions of users are willing to share their data to receive target ads, i.e., they do not want to block ads or trackers completely [3,4,5]
- In some other studies, a big crowd desires advanced methods to control their footprint [6,7]

[1] https://repository.upenn.edu/asc_papers/526
[2] https://doi.org/10.1016/j.ijhcs.2020.102498
[3] http://dl.acm.org/citation.cfm?doid=2162081.2162084
[4] https://www.usenix.org/conference/soups2015/proceedings/presentation/chanchary
[5] https://dl.acm.org/doi/10.1145/2335356.2335362
[6] https://dl.acm.org/doi/10.1145/2501604.2501612
[7] https://dl.acm.org/doi/10.1145/2501604.2501611

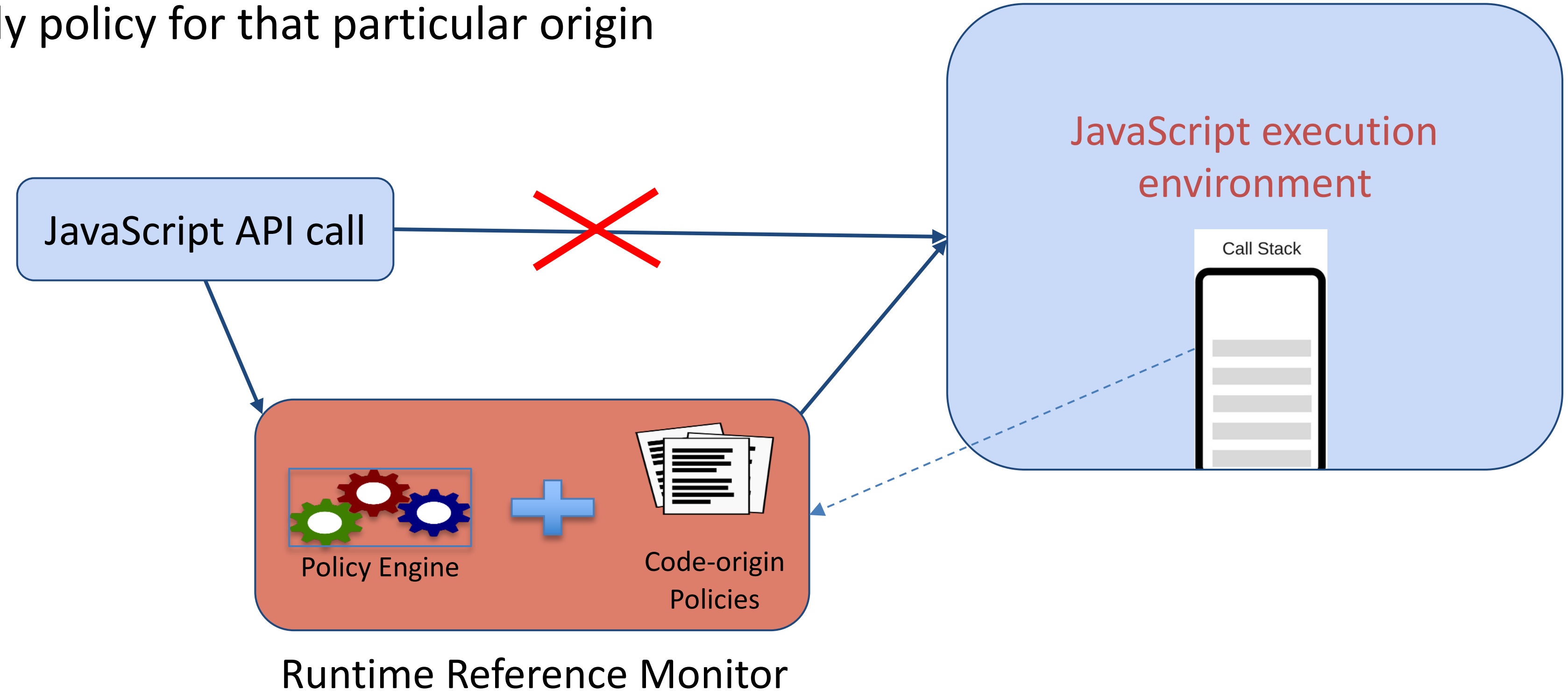
Our User-centric and Code-Origin Policy Approach

- Place a security reference monitor at runtime to mediate security and privacy relevant behaviors/actions
  - Trace the origin of the caller to actions/APIs, i.e., the code-origin
  - Basic policies as agreements/rules are defined by the developer/provider
    - Enforced at runtime and can be customized the end users
    - With formal assurance



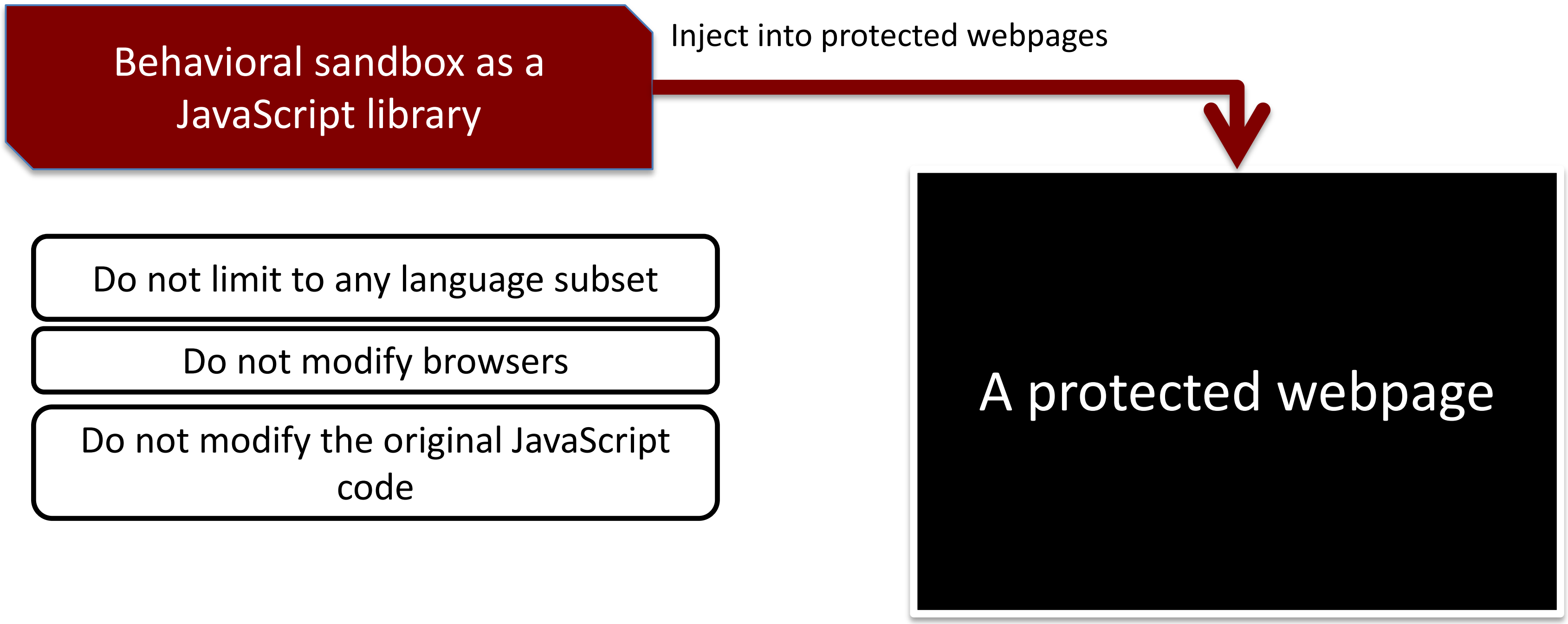
Code-Origin Runtime Reference Monitor

- Each relevant API call is wrapped with a monitor, based on the self-protecting JavaScript approach
  - Will check with the policy engine
    - Inspect the call stack for the origin of the code
      - Apply policy for that particular origin



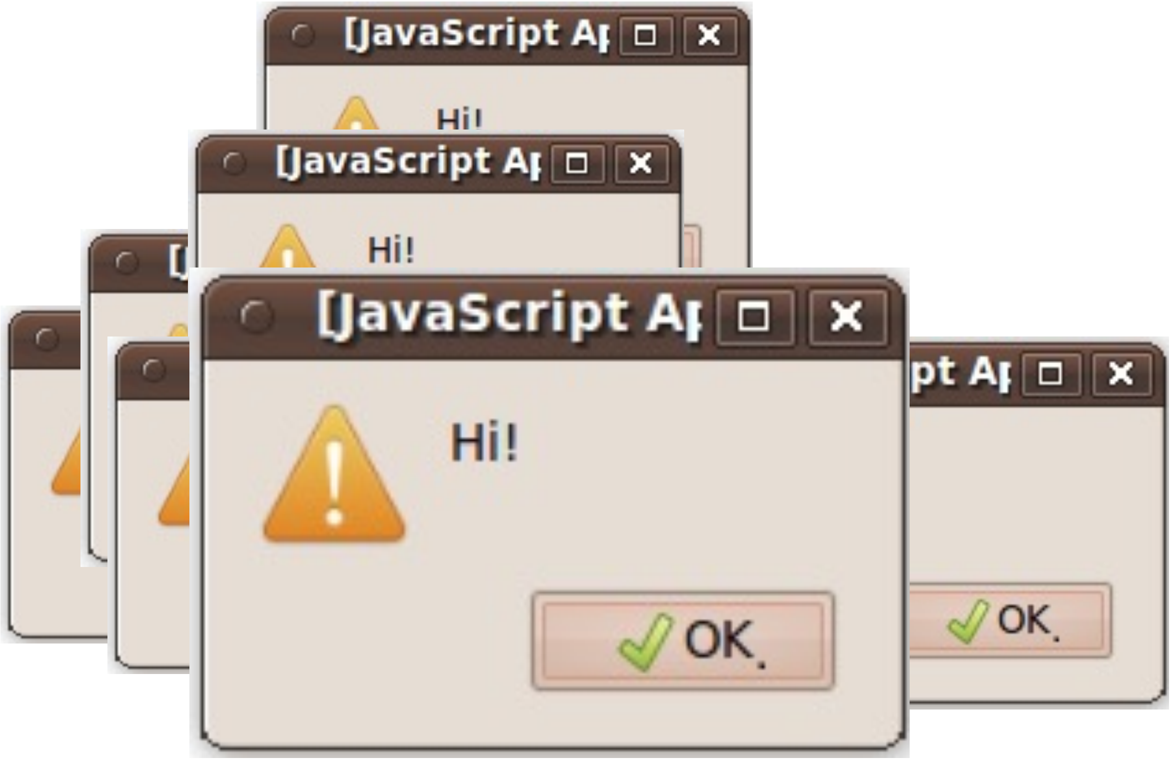
Lightweight Self-Protecting JavaScript [Phung et al., ASIACCS 2009]

- Provide a behavioral sandbox to control JavaScript execution



[Phung et al., ASIACCS 2009] Phung, P. H., Sands, D., and Chudnov, A., "Lightweight Self-protecting JavaScript," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS 2009, Sydney, Australia, pp. 47–60, ACM, March 2009. DOI: https://doi.org/10.1145/1533057.1533067

An Attack Example




```
window.alert('Hi!');
```

Challenges in JavaScript Security

- Code obfuscation

alert('Hi!');


```
var abcxyz = window.alert;
abcxyz('Hi!');
```

```
%61%6C%65%72%74%28%27%58%53%53%27%29%3B%0A%0A
```

Challenges in JavaScript Security

- Dynamic code generation

```
<script>
document.write('<scr');
document.write('ipt> malic');
var i= 1;
document.write('ious code; </sc');
document.write('ript>');
</script>
```

```
<script> malicious code; </script>
```

Wrapping security-relevant APIs

```
original_alert=window.alert;
window.alert = function(){
  if (policyCheck(..))
     execute(original_alert,..);
  else{..}
}
window.alert('Hi!');
```

1. Keep the original reference

2. Redefine the reference

3. Check policy to control the execution

Inject Self-Protecting JavaScript code before any other JavaScript code to monitor them

Self-Protecting JavaScript Deployment on Server-side

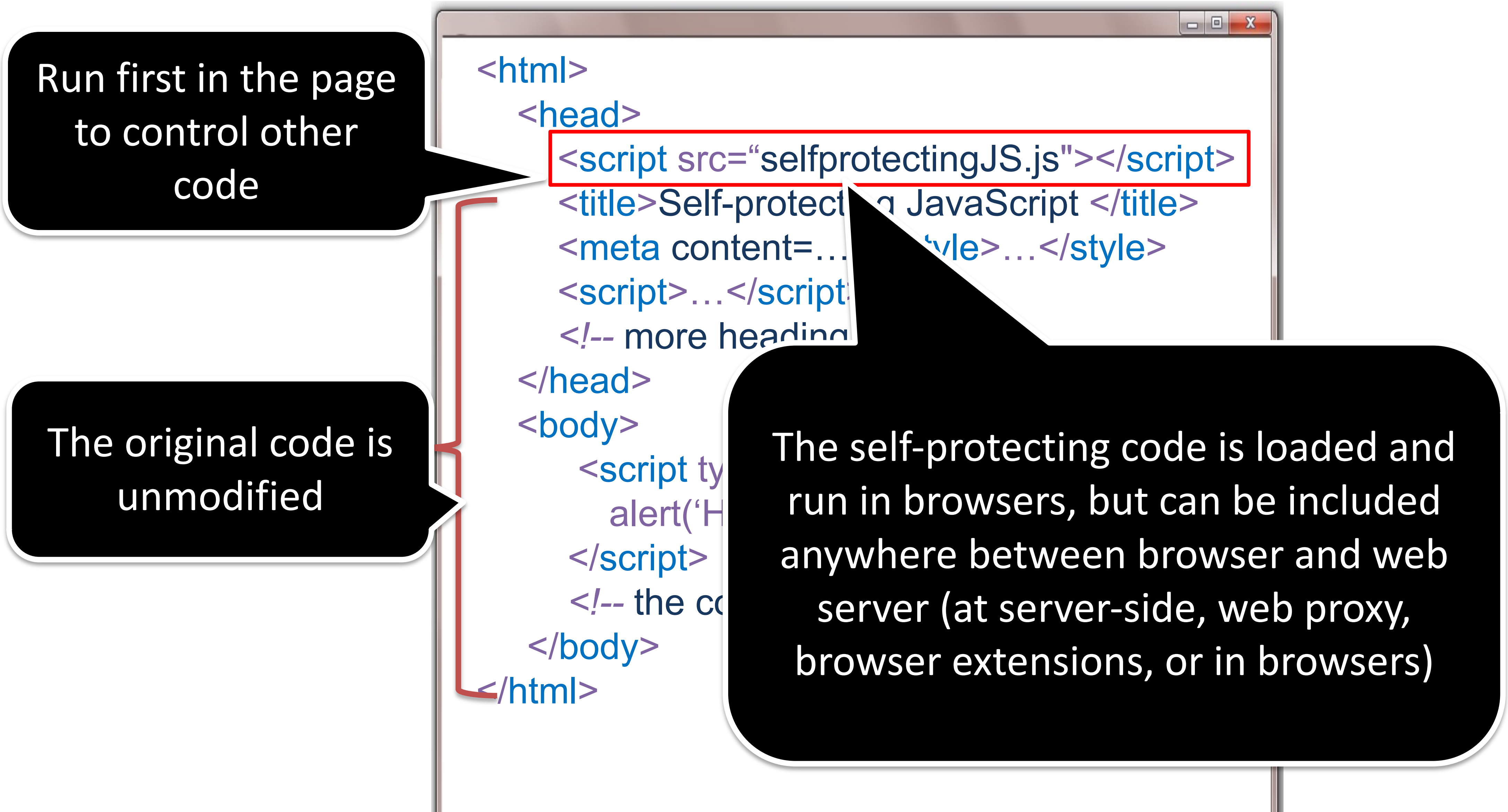
```
(function(){
    /*Self-Protecting JavaScript code
    within an anonymous function
    to protect itself from tamper-proofing */
})();
```

↓ Customized by the website owner

selfprotectingJS.js

↓ Included as the first script in the website

Self-Protecting JavaScript Deployment

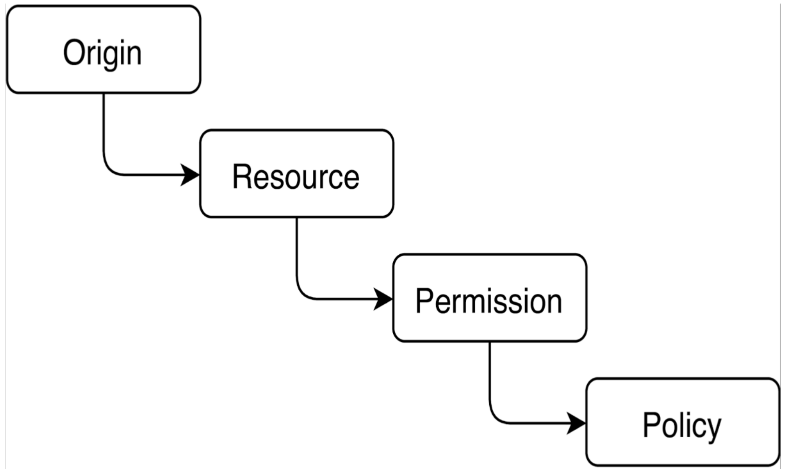


Self-Protecting JavaScript Summary

- Advantages
  - Can enforce runtime behavioral policies without modifying the browser or the original JavaScript code. Policy examples:
    - *`Limit the number of alerts to 2, of dynamic images to 1`*
    - *`Do not allow sending after reading sensitive information`*
    - *`Only allow links in a whitelist`*
- Limitations
  - Follow the same-origin policy, cannot distinguish where the actual code comes from
  - Depend on developers
    - End-users can only rely on developers
- Motivation:
  - How to define and enforce multiple party policies?

Code-Origin Policy Examples

- Grant access to APIs based on code-origin, e.g.,:
  - "trusted" code-origin can have full access to all resources
  - "local" code-origin l can have access to resources A, B
  - "remote1" code-origin can have access to resources C
  - "remote2" code-origin can have access to resources D
- More Fine-grained Policy Patterns
  - Resource bounds Policy
    - Limit the number of accesses to a resource
      - E.g.,: limit the number of Ajax request from a particular code-origin
  - Whitelist Policies
    - A resource access is allowed only under some conditions
      - E.g.,: allow data send to some predefined receipts
  - History-based Policies
    - Policies depending on the previous execution status
      - E.g.,: no sending after user data is read for a particular code-origin

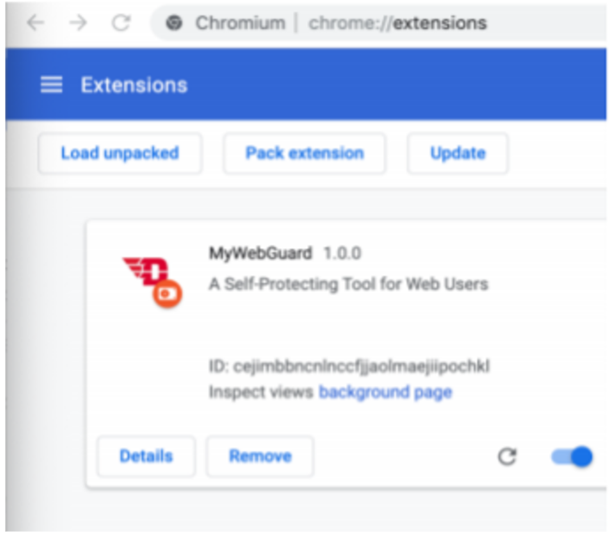



```
{
  "adservice" : {
    "location" : {
      "read" : {
        "enabled" : true
      }
    }
  }
}
```

User centric and Code-origin policies in Browsers

MyWebGuard [Hiremath et al., FDSE 2019, Phung et al., SNCS 2020]

- A mechanism at end-users side, e.g., in-browser or browser-extension
  - Can monitor JavaScript code behaviors
    - Enforce policies for each code origin, e.g., where the code come from
      - Do not need any new APIs



[Hiremath et al., FDSE 2019] Hiremath, P. N., Armentrout, J., Vu, S., Nguyen, T. N., Tran, M. Q., and Phung, P. H. (2019). MyWebGuard: Toward a User-Oriented Tool for Security and Privacy Protection on the Web. In *Proceedings of the 6th International Conference on Future Data and Security Engineering 2019* (FDSE 2019), volume 11814 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag.

[Phung et al., SNCS 2020] Phung, P. H., Pham. H. D., Armentrout, J., Panchakshari N. H. and Tran, M. Q.. "A User-Oriented Approach and Tool for Security and Privacy Protection on the Web." *SN Comput. Sci. 1 (2020): 222.*

MyWebGuard: code origin

- Use call stack at in the monitor (at runtime) to identify where a behavior comes from:

```
var callstack = new Error().stack;
var code_origin = getCodeOrigin(callstack);
```

- Enforce code origin-based policy for any websites
  - Allow or disallow an action based on
    - code origin
    - code behaviors
    - User choice

A Code-Origin Policy implementation example in MyWebGuard

- Monitoring cookie reading:

```
Object.defineProperty(document, "cookie", {
  get: function () { // monitor the cookie reading
    //.. security init code
    var callstack = new Error().stack;
    var code_origin = getCodeOrigin(callstack);
    if (originAllowed(code_origin,"document","cookie","get")) {
      //check the policy to see if the origin is allowed to read
      setOriginSourceRead(code_origin);
      return document_cookie_orginal_desc.get.call(document);
    }
    return;
  },
  set: function(val){ // monitor the cookie writing
    //policy for cookie writing
  },
  //security configuration
});
```

1. Monitor an action and get its real origin when the action is called

2. Check the policy Allow or disallow the action based on stateful policies

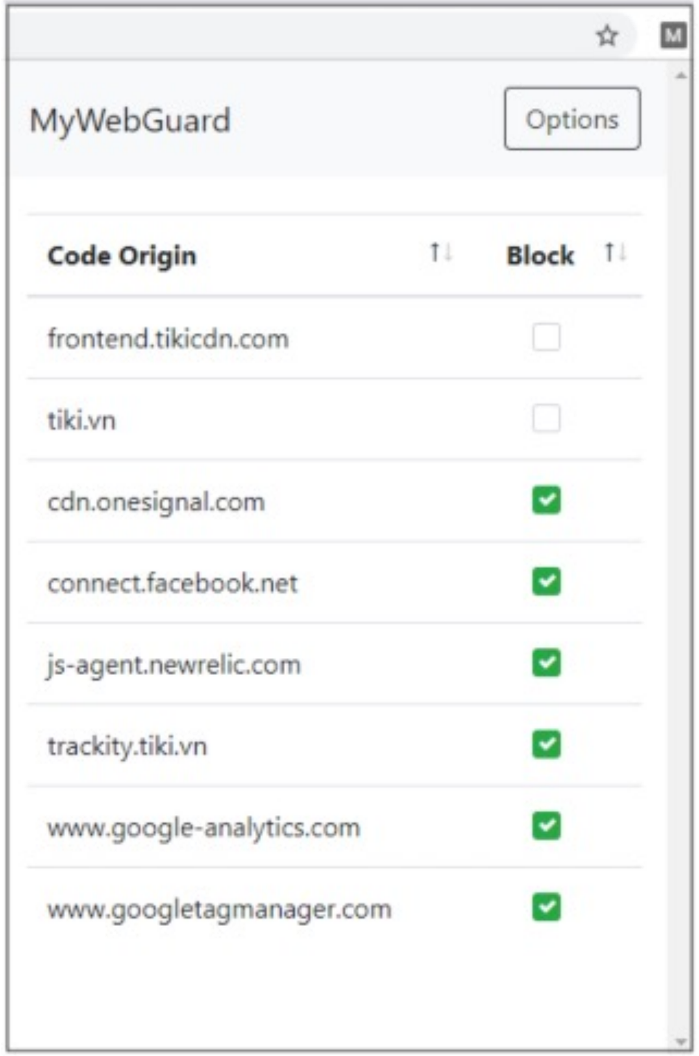
MyWebGuard Policy Examples

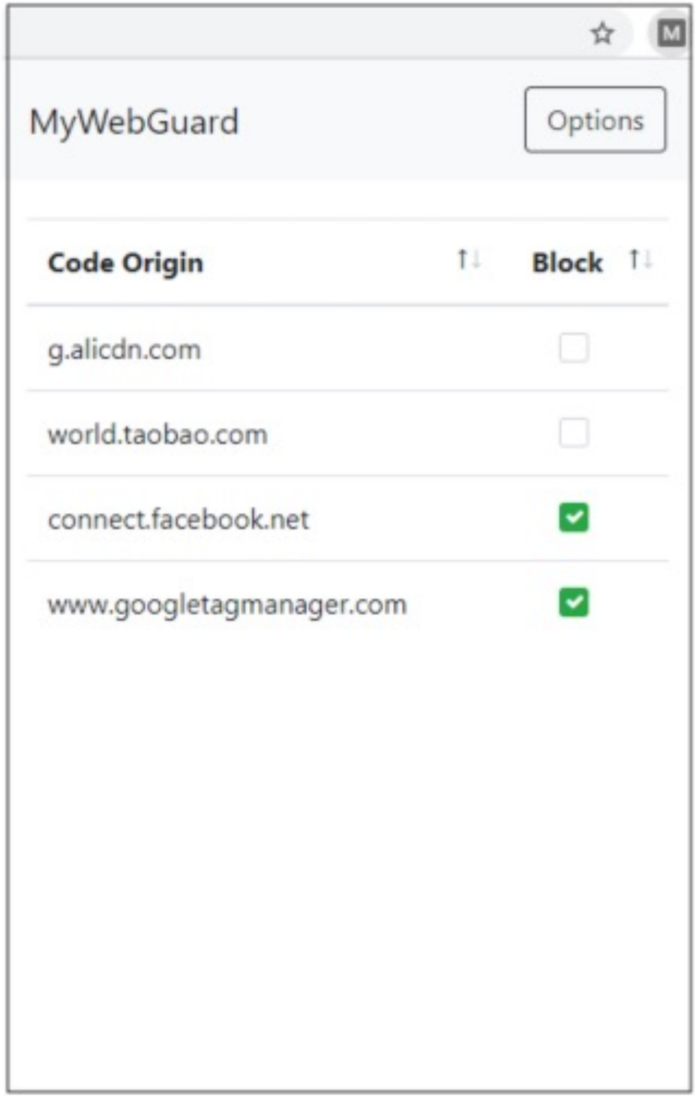
- Monitor and mark property read (data sources) for each code origin
    - `document.getElement*, localStorage.getItem, document.cookie, window.history, navigator.geolocation.getCurrentPosition` …

- Monitor data channels (sinks) sent from the browser
    - HTTP requests : Object of `Frame, IFrame, Image, Script, Form, Ajax, WebSocket`
        - General policy: no send after reading for each code origin
            - Ask users if needed

MyWebGuard User Interface

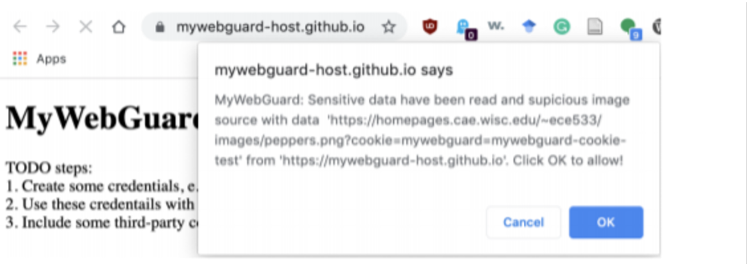
- Users can customize the policies further
  - Based on personal needs

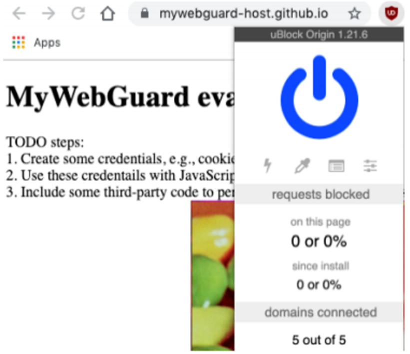


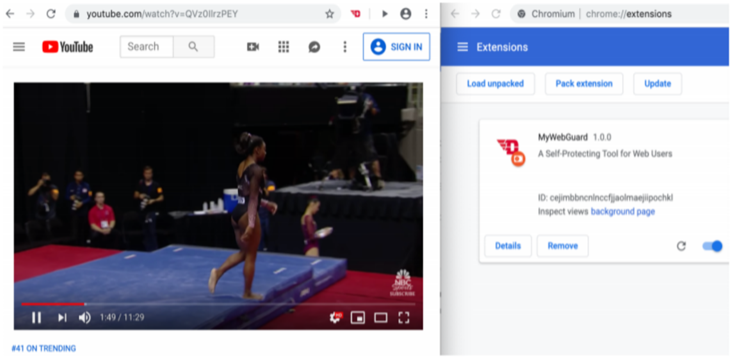


MyWebGuard Evaluation

- Can detect data/privacy leak channels
  - Leading tools, e.g., uBlock Origin, Ghostery or Brave browser ignore
- Allow users to decide if a suspicious action is detected but not defined in the leak channels
- Functional with popular websites

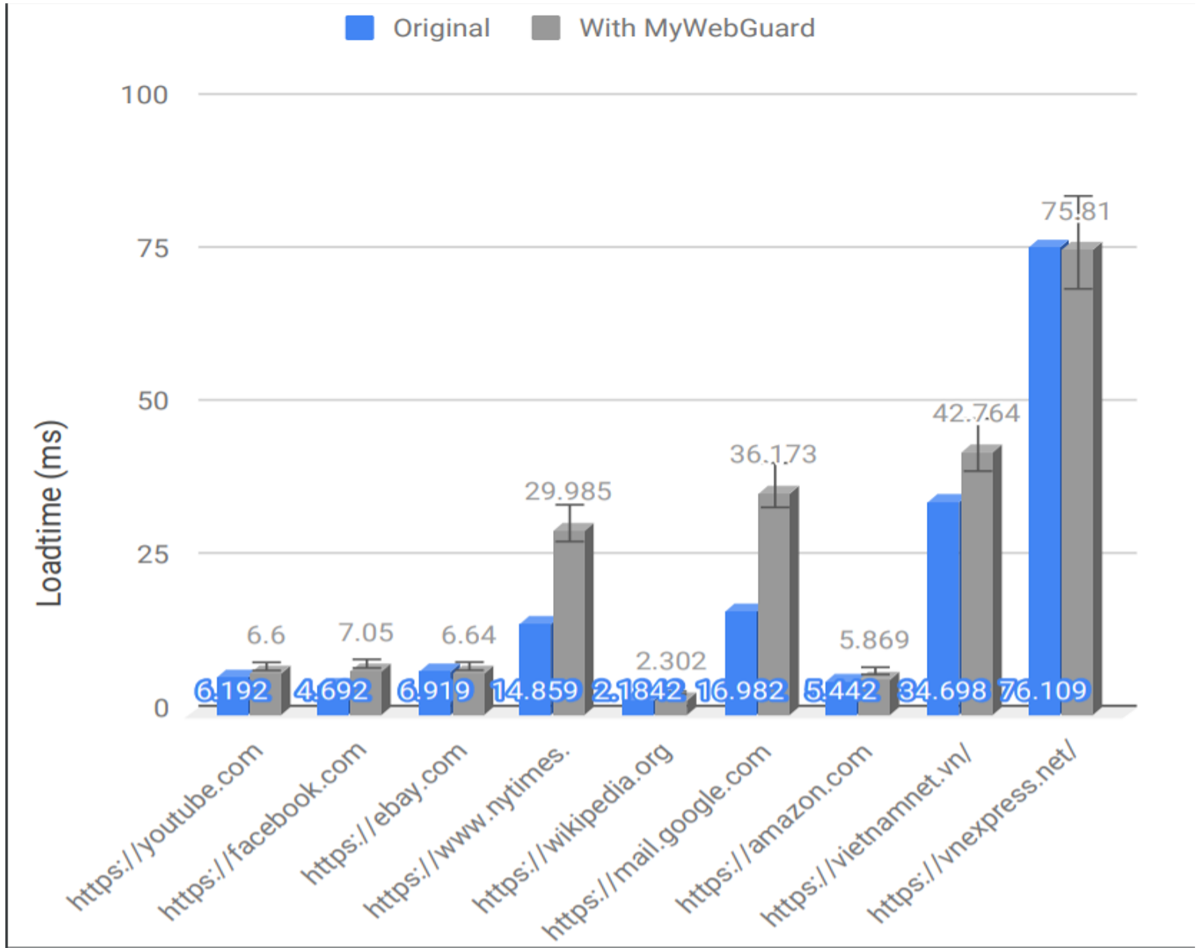




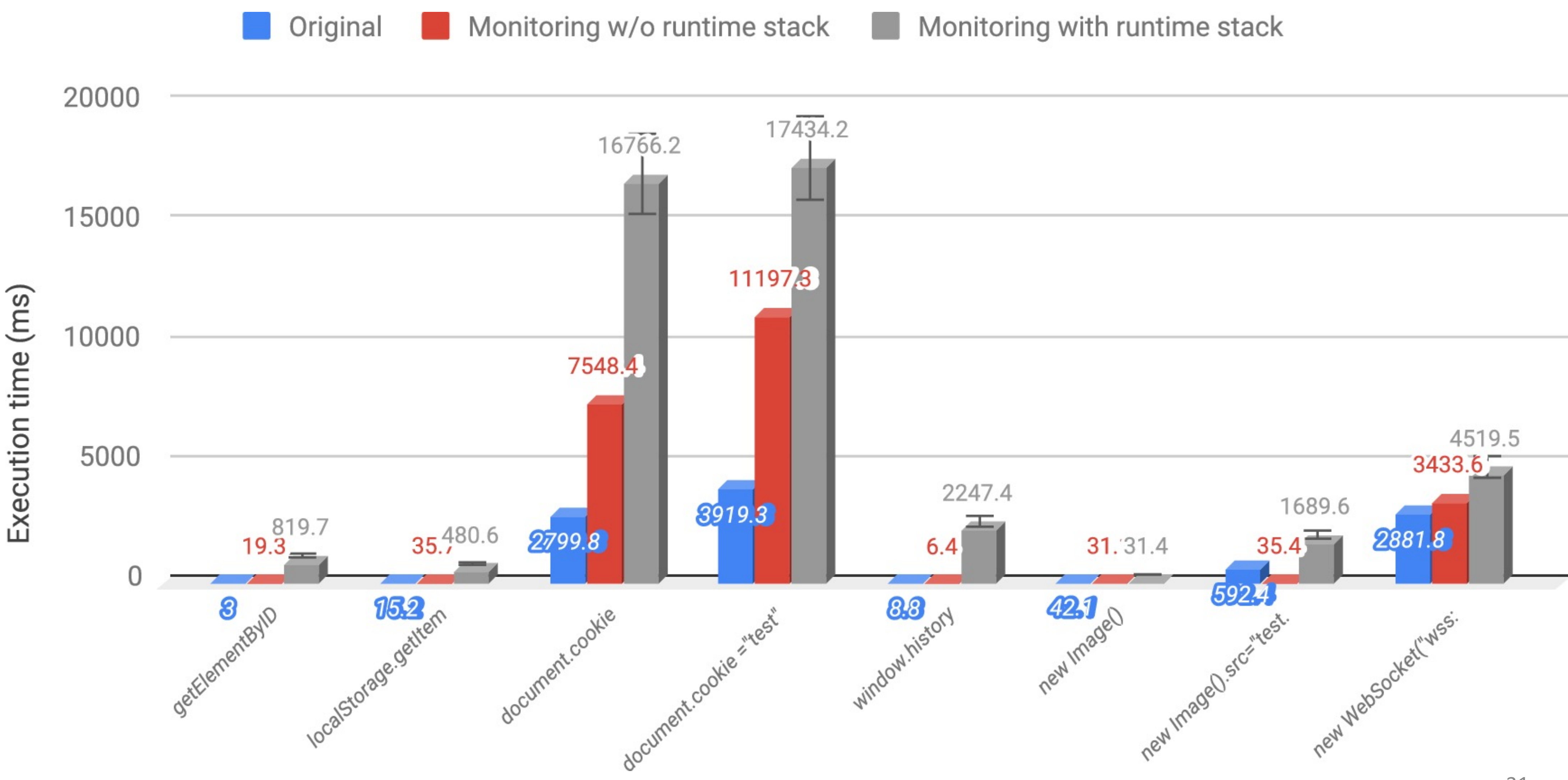


Runtime Evaluation

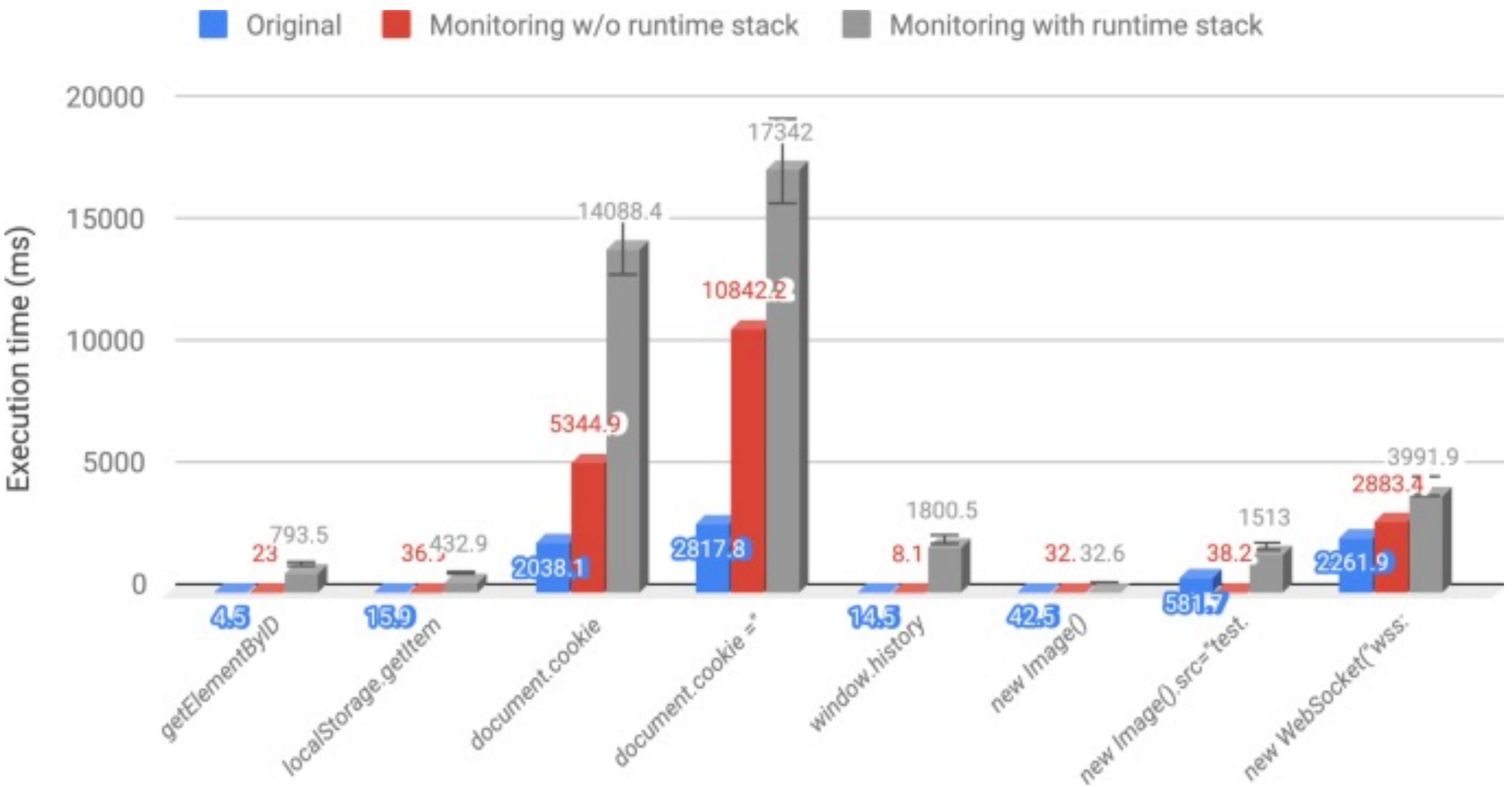
- We tested MyWebGuard with both Chromium and Brave browsers (on Ubuntu 18.04.2 LTS) on real websites
  - The overheads are not noticeable as shown in the graph



Microbenchmark of MyWebGuard on Chromium

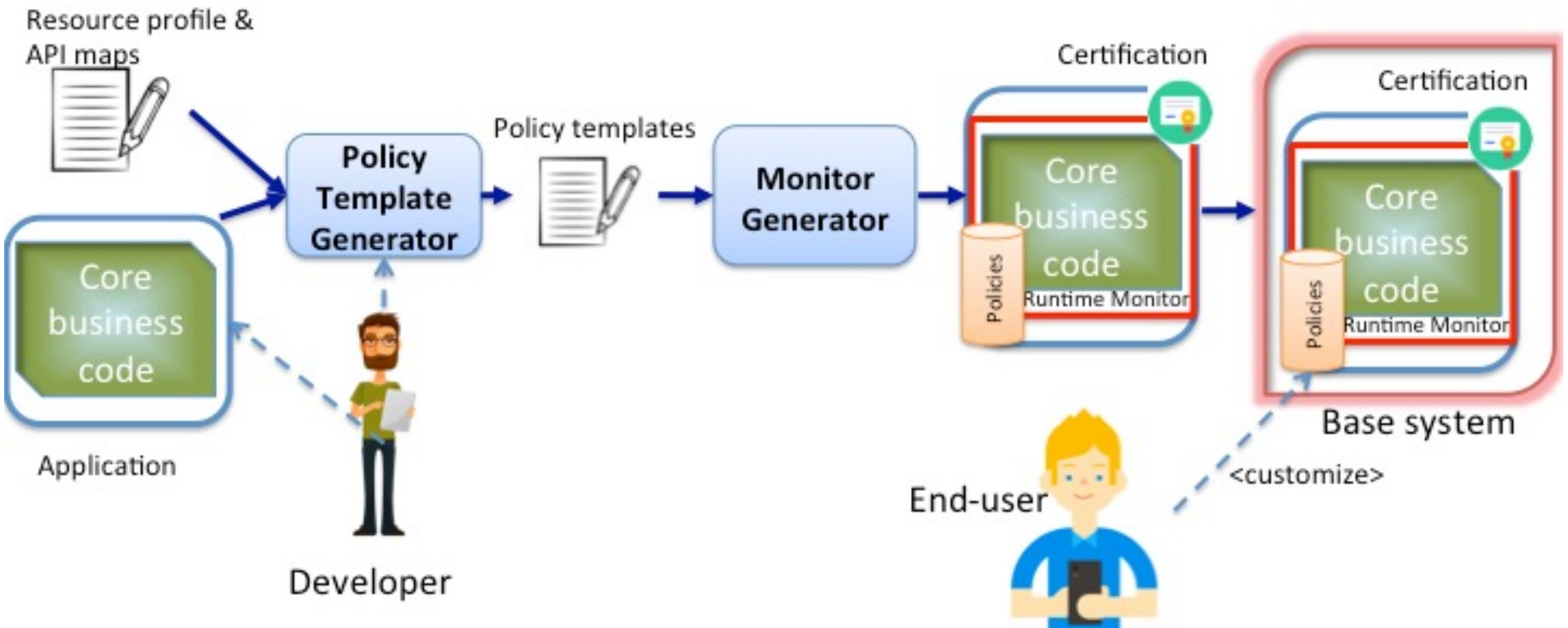


Microbenchmark of MyWebGuard on Brave



Code-Origin Policy Long-term vision

- Developers/Providers define formal privacy agreements in code-origin policy at the development phase
  - Tools will generate certificate together with code
    - The base system have a runtime monitor and verifier to provide assurance for policy enforcement
    -



The history and evolution of the Web

Source: Fabric Ventures

Code-Origin Policy

Open challenges

- Usability of code-origin policies
  - Need user studies and UX design
- Encode privacy regulations into code-origin policies
- Certificate generation and verification
- Integrate this code-origin policies and formal assurance into the browser

On-going and Future Work

- Student theses/work to be submitted for publications
  - Sunkaralakunta Venkatarama Reddy, Rakesh. *A User-Centric Security Policy Enforcement Framework for Hybrid Mobile Applications,* Master thesis, 2019. Online: http://rave.ohiolink.edu/etdc/view?acc_num=dayton1564744609523447
  - Rowland, Zachary S.. *A Study on Formal Verification for JavaScript Software*, Honors Thesis, 2021. Online: https://ecommons.udayton.edu/uhp_theses/334/
  - Nicholson, Timothy and Oei, James. A study of privacy laws and implementing them in MyWebGuard, Undergraduate Summer Research 2021
- Student thesis to be defended
  - Bishop, Douglas. *User-Centric Security and Privacy Protection In Browser.* Master thesis, expected to defend in December 2021.

Thank you

Phu H. Phung
Intelligent System Security Lab
Department of Computer Science
University of Dayton
https://isseclab-udayton.github.io
phu@udayton.edu

University of Dayton