



UNIVERSITY OF AMSTERDAM

Formally Defining Document Object Model Cross-site Scripting Exploits using Propositional Dynamic Logic

Research Project — Final Report

Student: Dirk van den Beuken (id: 16386051)
(dirk.vandenbeuken@gmail.com)
Supervisor: Dr. Malvin Gattinger
(b.r.m.gattinger@uva.nl)

MSc Security and Network Engineering
University of Amsterdam
June 30, 2026

Abstract

Cross-site scripting (XSS) is one of the most pervasive exploits in web security, yet formal studies of its most structural variant – Document Object Model Cross-site Scripting – remain limited. This thesis presents a formal framework for defining DOM XSS exploits in Propositional Dynamic Logic, approached from the attacker’s perspective. This model allows for reasoning about the conditions under which exploitation is possible.

The framework is built on a combined state model that integrates two components: (i) a DOM state space, which represents the structural properties of the document as a grove of uniquely identified nodes, and (ii) a JavaScript variable environment, tracking how attacker-controlled values are transformed during script execution. Program transitions across three semantic classes – source, flow, and sink – specify how a payload propagates from external input through intermediate data transformations to a point of execution in the DOM. Flow operations include assignment, concatenation, encoding, decoding, and sanitization, the last of which allows for reasoning about evasion techniques. An exploit state is defined as satisfying both a taint proposition and a sink node proposition, thereby clearly specifying when an attacker’s exploit has succeeded.

The validity of the formalization is demonstrated using three examples. The first example showcases an exploit amid a wrongfully applied sanitizer. The second example reveals a double-decode evasion with multiple execution pathways. Finally, the last example shows that a DOM clobbering attack cannot be captured by the formal model, thereby highlighting the limitations of the formalization.

Contents

1	Introduction	1
2	Related Work	2
3	Background	3
3.1	Cross-site Scripting	3
3.1.1	Reflected and Stored XSS	3
3.1.2	Document Object Model XSS	4
3.2	Logic Foundation	4
3.2.1	Propositional and First-order Logic	5
3.2.2	Modal Logic	5
3.2.3	Propositional Dynamic Logic	6
4	Conceptualizing DOM XSS	8
4.1	DOM as a State Space	8
4.2	JavaScript as a State Space	9
4.3	Combined State Model	11
5	Formalization in Propositional Dynamic Logic	13
5.1	Formal Definition of the Combined State	13
5.1.1	Atomic Propositions	13
5.1.2	Initial and Exploit States	14
5.2	Semantics of DOM and JavaScript Transitions	15
5.2.1	Transition Relations as Named Relations	15
5.2.2	Source Transition Rules	15
5.2.3	Flow Transition Rules	16
5.2.4	Sink Transition Rules	17
5.2.5	Taint-Propagation Constraints	18
6	Validation	20
6.1	Example One: Sanitization Wrongfully Applied	20
6.2	Example Two: Multiple pathways	23
6.3	Example Three: Formalization Limits	27
7	Discussion	29
8	Conclusion	30
	Appendix	32

1 Introduction

Web applications have evolved into complex, client-driven systems in which significant computational work takes place within the browser. While this enables highly interactive user experiences, it also exposes new attack possibilities. Cross-site scripting (XSS) attacks, in particular, exploit the dynamic foundation of client-side execution and continue to pose a serious security risk: keylogger-software can be inserted, sensitive data can be retrieved, and unauthorized API calls can be performed.

Cross-site scripting attacks are still one of the most prevalent threats. In fact, Acunetix's 2021 Web Vulnerability Report found that cross-site scripting was the fourth-highest severity vulnerability, which was detected in 25% of its scanned targets [Acc21]. Another study, conducted by Sebastian Lekies, Ben Stock, and Martin Johns investigated Document Object Model cross-site scripting (DOM XSS) specifically. They discovered that 6167 vulnerabilities were found across 5000 websites, thereby highlighting DOM XSS's significance even further [LSJ13].

Although many mitigation techniques have been built over the last decades, none of them conclusively eliminate DOM XSS. The same-origin policy restricts how scripts can interact with resources from different origins, but does not prevent the execution of injected scripts within a trusted context. Moreover, since this attack takes place within the browser, firewalls are usually circumvented. On the one hand, sanitization techniques provide a reasonably good defense against cross-site scripting attacks by filtering out special characters and strings, like scripting tags. On the other hand, formulating payloads in a different manner can still bypass these sanitization techniques too often.

To create more comprehensive rules, sanitization techniques have to be more precisely formulated and thereby more robust. That is, the rules have to reduce the number of false positives, while not increasing the number of false negatives. To formulate these rules, the mindset of an attacker performing DOM XSS attacks has to be understood first: an attacker has to manoeuvre in a wide variety of ways, depending on the website's structure, to find a workable exploit. This thesis aims to formalize and systematize the process by which DOM XSS exploits are constructed.

Propositional Dynamic Logic (PDL) is used as the means of formalization. Dynamic Logic is designed to reason about programs and their effects, where current properties are contained in states, and transitions between states are achieved through applying programs. Propositional Dynamic Logic is a foundational version, allowing for precise formal expressivity about the structural properties of DOM XSS exploits.

This leads to the research question:

"How can we formally define Document Object Model Cross-site Scripting Exploits using Propositional Dynamic Logic?"

The remainder of this thesis is structured as follows: The related work chapter outlines prior work on XSS formalization and relevant logic sources. Then, the background chapter introduces the necessary foundations in cross-site scripting and logic. Building on this, the conceptualization chapter develops a state space model of DOM XSS, which is subsequently encoded in PDL formalization. The validity of the formalization is demonstrated through a series of examples. The thesis closes with a discussion of the model's capabilities and limitations followed by a conclusion. Throughout this thesis, many symbols will be introduced. A list of those symbols can be found in the appendix.

2 Related Work

This thesis combines information from two largely separate fields: the security literature on cross-site scripting, and the formal literature on logic. This related work section contains the most relevant contributions from both fields to provide the groundwork for this thesis.

Early work by Melicher et al. introduced DOM XSS as a distinct and particularly subtle form of XSS, emphasizing that the vulnerability originates in client-side JavaScript and the manipulation of the DOM rather than in the server response itself [Mel+18]. This distinction is important for this thesis, because the attack does not merely concern injected data, but the way in which browser-side program execution transforms that data into unsafe behavior. Additionally, the empirical study by Lekies, Stock, and Johns provides an understanding of DOM XSS in practice [LSJ13]. By analyzing millions of DOM XSS flows across thousands of real-world websites, their work demonstrates the severe impact DOM XSS could have on websites. It also contains examples of DOM XSS, reasoning about the structural problems that could pave the way for DOM XSS exploits. One complementary paper discusses the current state of cross-site scripting [Hyd+15]. In particular, it investigates how often cross-site scripting exploits arise and outlines the state of scientific discoveries about XSS. More recently, Khodayari and Pellegrino provide a thorough treatment of DOM clobbering as a distinct and underappreciated attack technique, including an empirical analysis of its prevalence and a discussion of the structural conditions that make it possible [KP23]. Their work will be used to showcase the limitations of the formalization as constructed in this thesis.

On the logic side, the classical work *Propositional Dynamic Logic of Regular Programs* by Fischer and Ladner will be used as inspiration to construct the PDL-approach in this thesis [FL79]. This paper is central because it gives a formal logical framework in which programs can be treated as modal operators, making it possible to reason about what holds before and after program execution. Since the present thesis aims to represent DOM XSS exploits as Propositional Dynamic Logic formulas, this work provides the most direct theoretical foundation.

The textbook *Dynamic Logic* by David Harel, Dexter Kozen, and Jerzy Tiuryn will be used as the main reference [HKT00]. This book describes the foundation of logic in general and Propositional Dynamic Logic in particular, providing the formulations and definitions used throughout this thesis. Additionally, the textbook *Modal Logic* will be used as a second reference, since it contains definitions, descriptions and formulations from a different perspective [BRV02].

Another inspiration for this thesis is *Local Hoare Reasoning about DOM* by Gardner et al., which demonstrates that the Document Object Model can be treated as a local state space amenable to formal reasoning [Gar+08]. Their approach of representing the DOM as a collection of uniquely identified nodes organized into trees, forests, and groves is adopted in the conceptualization chapter of this thesis. The operational semantics of the JavaScript variable environment is inspired by Plotkin’s structural approach to operational semantics [Plo04], and the treatment of JavaScript states is informed by the operational semantics for JavaScript developed by Maffeis, Mitchell, and Taly [MMT08].

Taken together, these works establish both the significance and workings of DOM XSS alongside the formal approaches needed to reason about it in a precise manner. However, what the existing literature does not provide is a formalization that combines a DOM state space, JavaScript execution, and taint propagation into a single model capable of expressing exploits directly. Bridging that gap is the central contribution of this thesis.

3 Background

3.1 Cross-site Scripting

Cross-site Scripting (XSS) is a code injection attack where malicious JavaScript code is inserted into a website. Cross-site scripting is executed on the client's browser, such that injected scripts are only executed on the client's browser while the webserver is not directly involved in the exploit. JavaScript is utilized as the scripting-language that incorporates interactivity within websites, as it transforms webpages from static to interactive. For example, JavaScript allows you to click links, validate user inputs, and use Application Programming Interfaces (APIs). If a vulnerability is present, JavaScript code from the client can be executed by the browser, thereby playing into the website's mishandling of user input. By performing an XSS attack, the attacker can consequently take advantage of these dynamic features and insert keyloggers, steal cookies or utilize APIs. Cybersecurity professionals usually test if they can exploit an apparent XSS vulnerability by inserting an `alert()` function, because it provides a quick and easy way to showcase that the website is actually executing your JavaScript payload, instead of merely reflecting a string back.

There are three types of cross-site scripting: (i) Reflected XSS, (ii) Stored XSS, and (iii) Document Object Model XSS, all of which will be discussed below.

3.1.1 Reflected and Stored XSS

Reflected cross-site scripting takes place if the website receives malicious code from the attacker, and immediately sends it back (i.e. it "reflects" the payload). The script used in reflected XSS does not reside on the server, hence this attack is not persistent [Hyd+15]. Even though this attack is non-persistent, a lot of harm can be invoked: An attacker could still request the server to retrieve sensitive information and reflect it back to the attacker. Reflected XSS is also called Non-Persistent XSS or Type-I XSS. Listed below is a typical example of a reflected XSS vulnerability in search bars, where the inserted string `name` is directly passed on to an `innerHTML`. An HTML string exploit containing inline JavaScript code, such as `<img src=x onerror=alert(1)>`, demonstrates the reflected XSS vulnerability.

```
1 const params = new URLSearchParams(window.location.search);
2 const name = params.get("name");
3
4 document.getElementById("output").innerHTML =
5     "Hello " + name;
```

Stored cross-site scripting, as its name implies, injects malicious code in a way that makes the website store the payload. So instead of just reflecting the payload back, the payload is permanently stored in a backend database system [Hyd+15]. This could be done on comment sections, logs, etc. When a victim visits the webpage where the payload is stored, the exploit will be executed. Stored XSS is also called Persistent XSS or Type-II XSS. An example of a comment section prone to stored XSS attacks can be seen below, which is vulnerable to the same exploit as the reflected XSS example:

```
1 // Storing comment without sanitization
2 app.post("/comment", (req, res) => {
3     database.save(req.body.comment);
4 });
5
6 document.getElementById("comments").innerHTML = storedComment;
```

Both reflected and stored cross-site scripting vulnerabilities can be found all across websites, given that the input is not properly sanitized.

3.1.2 Document Object Model XSS

Document Object Model cross-site scripting is a type of XSS where the exploit takes place in browser's Document Object Model (DOM). The *Document Object Model* is a programming interface for webpages, as it represents the page so that programs can change the document structure. The Document Object Model connects webpages to scripts by providing the structure of the document. Within the DOM, content is expressed in a logical tree structure, where the building blocks are called "nodes". Because of this structural implementation, the server is not involved in the exploit, which makes it difficult for servers to infer that a new exploit has been performed. DOM XSS is different from reflected or stored XSS, since the server is not involved and because the exploit has to be tailored in such a way that it can manoeuvre through the DOM before the actual exploit takes place [Mel+18]. DOM XSS has two very important parts: the source and the sink. The *source* is the place through which the payload is inserted. Examples include `document.referrer` and `location.hash`. The *sink*, contrarily, is the place where the payload is executed. This also contains many options, among them `eval()` and `document.write()`. As listed below, a DOM XSS attack can take place using the source `location.hash` to insert JavaScript code, and the sink `innerHTML` to execute that code.

```
1 const hash = window.location.hash.substring(1);
2
3 document.getElementById("content").innerHTML = hash;
```

An attack payload, for this exploit, can in principle be any HTML code which ultimately leads to the execution of JavaScript code, such as:

`https://example.com/#<img src=x onerror=alert(1)>`

3.2 Logic Foundation

Logic is the formal study of reasoning based on symbolic systems. A logic consists of three components: a language, a semantics, and a deductive system [HKT00, p. 67].

- A *language* defines a syntax. It is a set of rules that dictates which formulas are in accordance with logic formation rules, and can therefore be described as "well-formed formulas" (wff). A language consists of an alphabet and grammar. The *alphabet* is a set of symbols consisting of variables ($p, q, r, \dots$), *logical connectives* ($\neg, \wedge, \vee, \rightarrow$) and *punctuation* (parentheses). The *grammar* specifies how symbols should be combined. The example grammar $\varphi ::= p \mid \varphi \wedge \varphi$ allows formula $p \wedge p$ to be derived. A language is not concerned with whether the statement is "true" or "false": It only cares about the legality of the expressions [BV07].
- A *semantics* specifies how expressions should be interpreted as statements. It assigns truth values to expressions, denoting if an expression is "true" or "false" in a given model. This is formalized using a satisfaction relation $\models$ between models M and formulas φ , written $M \models \varphi$ [BV07].
- A *deductive system*, also called a proof system, allows the derivation of new formulas from existing formulas using mechanical rules. These derivations can be applied without any knowledge of the statements themselves. A deductive system D is a pair (A, R) , where $A \subseteq L$ is a set of logical axioms and R is a set of inference rules that specify how formulas can be derived from other formulas. A set of premises Γ can lead to a derivation of φ , written $\Gamma \vdash \varphi$ [HC96]. In this thesis, mainly semantics will be used as opposed to proof systems.

There are different logic systems, ranging from trivial to complex. Below, an outline of logic systems is given in hierarchical order, leading up to Propositional Dynamic Logic.

3.2.1 Propositional and First-order Logic

Propositional Logic Propositional Logic is the most basic type of logic, since it solely provides the possibility to formulate simple statements. For example, the statement "if you encounter a cat, you pet it", can be defined as $\varphi \rightarrow \psi$ for cat encounter φ and petting ψ . In between meta-variables, such as φ and ψ , formulas can be constructed using *operators*. The propositional operators used as connectives are $\wedge$ (conjunction), $\vee$ (disjunction) and $\neg$ (negation), together with the constants 1 (truth) and 0 (falsity), and the implication $\rightarrow$ and equivalence $\leftrightarrow$ [HKT00, p. 71]. These propositional statements can be useful for describing static phenomena. However, its applicability is quite constrained. Using Propositional Logic, sets cannot be described.

First-order Logic *First-order Logic*, also called predicate logic, is an extension of Propositional Logic and can be used to reason about sets. To achieve this, it introduces two new operators: the existential quantifier $\exists$ ("there exists") and the universal quantifier $\forall$ ("for all"). These quantifiers are used together with variables and predicates. Let $P(x)$ denote " x is a cat" and $Q(x)$ denote " x is petted". Then the formula $\forall x(P(x) \rightarrow Q(x))$ means that all cats are petted. Formulas in First-order Logic can be made more complex, resulting in statements like:

$$\forall x(P(x) \rightarrow \exists y(C(y) \wedge (E(x, y) \wedge (Q(x, y) \wedge H(y)))))$$

For all persons $P(x)$, there exists a cat $C(y)$ and there is an encounter between the person and cat $E(x, y)$ such that the person is petting the cat $Q(x, y)$ and the cat is happy $H(y)$.

3.2.2 Modal Logic

Modal Logic is a different approach from First-order Logic, encompassing reasoning about possibility and necessity, in addition to "true" or "false": As in Propositional Logic and First-order Logic, formulas are ultimately true or false, but their truth is evaluated relative to a state (or world) in a model. Modal Logic uses operators $\Diamond$ and $\Box$ for possibility and necessity respectively. Using these Modal Logic operators, we can formulate statements like (i) $\Diamond\varphi$ and (ii) $\Box\varphi$ to signify that (i) φ possibly holds and that (ii) φ necessarily holds. These statements can be written in other equivalent forms: $\Box\varphi \leftrightarrow \neg\Diamond\neg\varphi$ [BRV02].

Kripke Semantics *Kripke Semantics* is used to interpret modal formulas. Contrary to using declarations describing statements with words like "necessity" and "possibility", Kripke semantics makes use of so called "Worlds". So instead of stating "necessary φ ", we can say: "In every world, φ is true". Likewise, "possibly φ " can be formulated as: "There is at least one world in which φ is true". This way of approaching statements can express its semantics more starkly.

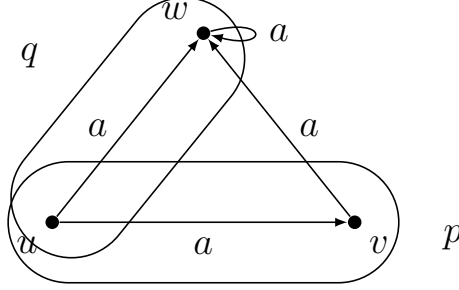
Kripke Frames and Models In Modal Logic, a *Kripke Frame* is a structure $\mathfrak{F} = (K, R_{\mathfrak{F}})$, where K is the universe of $\mathfrak{F}$ such that the elements of K can be called *states* or *worlds* [HKT00]. Using $R_{\mathfrak{F}}$, the relations between these states are specified, indicating which worlds are reachable from a given world. As an example, reaching world w from world v is possible, in case $(v, w) \in R_{\mathfrak{F}}$ holds. Kripke Frames can be extended to Kripke Models. A *Kripke Model* is a structure $\mathfrak{M} = (K, R_{\mathfrak{M}}, \mathbf{m}_{\mathfrak{M}})$ that also contains valuation function $\mathbf{m}_{\mathfrak{M}}$. The *valuation function* determines the truth assignment to the primitive propositions in every state, denoted as $\mathfrak{M}, w \models v \Leftrightarrow w \in \mathbf{m}_{\mathfrak{M}}(v)$. This is inductively extended to all modal formulas according to the following rules [HKT00, p. 128]:

$$\begin{aligned} \mathbf{m}_{\mathfrak{M}}(\varphi \rightarrow \psi) &\stackrel{\text{def}}{=} (K - \mathbf{m}_{\mathfrak{M}}(\varphi)) \cup \mathbf{m}_{\mathfrak{M}}(\psi), \\ \mathbf{m}_{\mathfrak{M}}(0) &\stackrel{\text{def}}{=} \emptyset, \\ \mathbf{m}_{\mathfrak{M}}(\Box\varphi) &\stackrel{\text{def}}{=} K - R_{\mathfrak{M}}^{-1}[K - \mathbf{m}_{\mathfrak{M}}(\varphi)], \end{aligned}$$

Resulting in:

$$\mathbf{m}_{\mathfrak{M}}(\Diamond\varphi) = R_{\mathfrak{M}}^{-1}[\mathbf{m}_{\mathfrak{M}}(\varphi)].$$

Example Let's say we have a Kripke model $\mathfrak{K} = (K, R_{\mathfrak{K}}, \mathbf{m}_{\mathfrak{K}})$, where $K = \{u, v, w\}$, $R_{\mathfrak{K}} = \{(u, v), (u, w), (v, w), (w, w)\}$ and proposition $q \in w, u$ and $p \in u, v$. We can display that in a diagram, as shown below.



Formulation The statements above are given in *set-theoretic* formulation. In this semantics, each formula φ is interpreted as a set of worlds $\mathbf{m}_{\mathfrak{K}}(\varphi) \subseteq K$, which allows for an adequate treatment of relational structure and composition. The *satisfaction relation* is introduced as an equivalent but secondary notation [BRV02]:

$$\mathfrak{K}, w \models \varphi \quad \Leftrightarrow \quad w \in \mathbf{m}_{\mathfrak{K}}(\varphi).$$

Throughout this thesis, the set-theoretic semantics $\mathbf{m}_{\mathfrak{K}}(\varphi)$ will be the main formulation and is used when defining or manipulating semantic operators (e.g., relational composition, modal operators, or program semantics). The satisfaction relation $\mathfrak{K}, w \models \varphi$ will sometimes be used, namely, when stating properties at the level of individual worlds, particularly for readability or when expressing logical consequences.

3.2.3 Propositional Dynamic Logic

Dynamic Logic is a branch of Modal Logic, specifically tailored to reason about programs and their applications. Propositional Dynamic Logic (PDL) in particular is the most fundamental form of Dynamic Logic. Propositional Dynamic Logic can describe formal properties of computer programs and utilize them within logical statements. Using PDL, we can formally reason about state transitions: we can determine whether a proposition holds in a final state after a specific program terminates, or whether a valid execution path exists.

The statement $[\alpha]\varphi$ states that φ is necessarily true after executing program α . To signify that φ is possible after executing program α , statement $\langle\alpha\rangle\varphi$ is used. One program can be applied to a conjunction of propositions: $[\alpha](\varphi \wedge \psi) \leftrightarrow [\alpha]\varphi \wedge [\alpha]\psi$. A sequence of programs, also called *complex programs*, can be noted as well:

$$[\alpha; \beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$$

Formulas, Programs and Symbols Propositional Dynamic Logic utilizes both *formulas* $\varphi, \psi, \dots$ and *programs* $\alpha, \beta, \gamma, \dots$. *Atomic programs*, i.e. the programs that only perform one operation, are denoted as $a, b, \dots$, creating the set of atomic programs Π_0 . Atomic propositions, however, are denoted $p, q, r, \dots$, creating the set of atomic propositions Φ_0 . Sets Π_0 and Φ_0 are subsets of the set of all programs Π and the set of all propositions Φ respectively. Common propositional operators are $\rightarrow$ for implication and $\perp$ for falsity. Program operators are $;$, $\cup$ and $*$, to denote composition, choice and iteration respectively. Finally, the test operator $?$ is very useful in the context of modeling programs, as it signifies that the subsequent statement should be executed in case a certain proposition holds: $\varphi?$ states that you should proceed if φ is true, else abort [HKT00].

Syntax The syntax of Propositional Dynamic Logic is defined using Backus-Naur Form (BNF). Note that the definitions of programs α and formulas φ are mutually dependent: programs may contain formulas via the test operator $\varphi?$, while formulas may contain programs via the modal operator $[\alpha]\varphi$. Therefore, they are defined simultaneously. The sets of programs α and formulas φ are given by:

$$\begin{aligned}\alpha &::= a \mid \alpha; \alpha \mid \alpha \cup \alpha \mid \alpha^* \mid \varphi? \\ \varphi &::= p \mid 0 \mid \varphi \rightarrow \varphi \mid [\alpha]\varphi\end{aligned}$$

The *dual modality* is defined as $\langle \alpha \rangle \varphi := \neg[\alpha]\neg\varphi$.

Computer Programs PDL is designed to describe and define computer programs, since program code can be precisely expressed in PDL statements [HKT00, p. 167]:

$$\begin{aligned}\mathbf{skip} &\stackrel{\text{def}}{=} 1? \\ \mathbf{fail} &\stackrel{\text{def}}{=} 0? \\ \mathbf{if} \varphi_1 \rightarrow \alpha_1 \mid \dots \mid \varphi_n \rightarrow \alpha_n \mathbf{fi} &\stackrel{\text{def}}{=} \varphi_1?; \alpha_1 \cup \dots \cup \varphi_n?; \alpha_n \\ \mathbf{do} \varphi_1 \rightarrow \alpha_1 \mid \dots \mid \varphi_n \rightarrow \alpha_n \mathbf{od} &\stackrel{\text{def}}{=} (\varphi_1?; \alpha_1 \cup \dots \cup \varphi_n?; \alpha_n)^*; (\neg\varphi_1 \wedge \dots \wedge \neg\varphi_n)? \\ \mathbf{if} \varphi \mathbf{then} \alpha \mathbf{else} \beta &\stackrel{\text{def}}{=} \varphi?; \alpha \cup \neg\varphi?; \beta \\ \mathbf{while} \varphi \mathbf{do} \alpha &\stackrel{\text{def}}{=} (\varphi?; \alpha)^*; \neg\varphi? \\ \mathbf{repeat} \alpha \mathbf{until} \varphi &\stackrel{\text{def}}{=} \alpha; (\neg\varphi?; \alpha)^*; \varphi?\end{aligned}$$

Semantics While the aforementioned definitions provide a syntactic way to express program structures, they require a formal semantic interpretation to determine validity. In PDL, this is achieved by extending Kripke semantics to account for both propositions and programs. A PDL structure $\mathfrak{K} = (K, \mathbf{m}_{\mathfrak{K}})$ consists of a set of states K and a valuation function $\mathbf{m}_{\mathfrak{K}}$ that maps atomic propositions to sets of states and atomic programs to relations on K . The semantics of complex formulas and programs are then defined by mutual induction as follows [HKT00, p. 168]:

$$\begin{aligned}\mathbf{m}_{\mathfrak{K}}(\varphi \rightarrow \psi) &\stackrel{\text{def}}{=} (K - \mathbf{m}_{\mathfrak{K}}(\varphi)) \cup \mathbf{m}_{\mathfrak{K}}(\psi) \\ \mathbf{m}_{\mathfrak{K}}(\neg\varphi) &\stackrel{\text{def}}{=} K - \mathbf{m}_{\mathfrak{K}}(\varphi) \\ \mathbf{m}_{\mathfrak{K}}(0) &\stackrel{\text{def}}{=} \emptyset \\ \mathbf{m}_{\mathfrak{K}}([\alpha]\varphi) &\stackrel{\text{def}}{=} K - (\mathbf{m}_{\mathfrak{K}}(\alpha)^{-1}[K - \mathbf{m}_{\mathfrak{K}}(\varphi)]) \\ &= \{u \mid \forall v \in K \text{ if } (u, v) \in \mathbf{m}_{\mathfrak{K}}(\alpha) \text{ then } v \in \mathbf{m}_{\mathfrak{K}}(\varphi)\} \\ \mathbf{m}_{\mathfrak{K}}(\alpha; \beta) &\stackrel{\text{def}}{=} \mathbf{m}_{\mathfrak{K}}(\alpha) \circ \mathbf{m}_{\mathfrak{K}}(\beta) \\ &= \{(u, v) \mid \exists w \in K (u, w) \in \mathbf{m}_{\mathfrak{K}}(\alpha) \text{ and } (w, v) \in \mathbf{m}_{\mathfrak{K}}(\beta)\} \\ \mathbf{m}_{\mathfrak{K}}(\alpha \cup \beta) &\stackrel{\text{def}}{=} \mathbf{m}_{\mathfrak{K}}(\alpha) \cup \mathbf{m}_{\mathfrak{K}}(\beta) \\ \mathbf{m}_{\mathfrak{K}}(\alpha^*) &\stackrel{\text{def}}{=} \mathbf{m}_{\mathfrak{K}}(\alpha)^* = \bigcup_{n \geq 0} \mathbf{m}_{\mathfrak{K}}(\alpha)^n \\ \mathbf{m}_{\mathfrak{K}}(\varphi?) &\stackrel{\text{def}}{=} \{(u, u) \mid u \in \mathbf{m}_{\mathfrak{K}}(\varphi)\}\end{aligned}$$

4 Conceptualizing DOM XSS

To express DOM XSS exploits in Propositional Dynamic Logic, a conceptual framework has to be designed first. A DOM XSS exploit is a *process*, which needs to be defined as a series of propagation operations through the Document Object Model and the JavaScript code. These propagation operations can also be viewed as the manner in which an exploit should be constructed, consisting of state transitions across both the JavaScript environment and the DOM. In this section, a conceptual build-up is created as a template for the next chapter, where this conceptual build will be encoded in PDL.

4.1 DOM as a State Space

To formalize DOM XSS exploits, a definition of the environment in which these exploits operate has to be created first. This definition will mainly be derived from the paper *Local Hoare Reasoning about DOM*, where the DOM is modelled as a local state space: instead of representing the full Document Object Model, only the relevant part of the DOM is defined [Gar+08]. This local approach is particularly suited for XSS analysis, as it allows us to reason about the propagation from source to sink without requiring a complete representation of the entire Document Object Model.

Tree structure and Identifiers In this model, the DOM is not merely a hierarchy but a collection of nodes that can be uniquely identified. Every element and every list of children gets a unique identifier assigned. For XSS research, these identifiers are essential to track how malicious data propagates to a specific node in the DOM.

Formal Data Structures The state space is defined using four core data types $D \in \{T, F, G, S\}$, which allow for a recursive representation of the DOM:

- **Trees** ($t \in T$): Represent individual nodes. A node is either an element $s_{id}[A, f]_{fid}$ (with a tag s , a node ID, a set of attributes A , a child forest f , and a forest ID) or a text node $\#text_{id}[s]$. Trees are fundamental, because they arrange the order through which the exploit could propagate.
- **Forests** ($f \in F$): Ordered lists of trees, representing the children of an element. Forests allows for treatment of the children from the root as separate pathways through which the exploit could propagate.
- **Groves** ($g \in G$): The "object store" or heap, containing the set of all top-level trees. In an XSS context, the Grove contains both the active document and any "in-memory" fragments created by a script. Using groves, we can separate the elements that are currently applicable and make changes before influencing the rest of the DOM elements.
- **Strings** ($s \in S$): The actual data, which is often the vector for XSS payloads. Changing the string values, we can change the input field and discover which string injections result in exploitation.

The following recursive definition allows for the treatment of a vulnerable node as a local tree that can be extracted from or inserted into a larger grove using logic. The grammar for these structures is defined as follows [Gar+08]:

$$\begin{aligned}
 t &::= s_{id}[A, f]_{fid} \mid \#text_{id}[s] \\
 f &::= \emptyset_F \mid \langle t \rangle_F \mid f \otimes f \\
 g &::= \emptyset_G \mid \langle t \rangle_G \mid g \oplus g \\
 s &::= \emptyset_S \mid c \mid s \cdot s
 \end{aligned}$$

Each element node carries an attribute map $A : \mathcal{A} \rightarrow S$, a partial function assigning string values to attribute names drawn from the universe of attribute names $\mathcal{A}$. Writing $A(a) = s$ means the node has attribute a set to value s . Attributes absent from the map are undefined. The empty map is written $\emptyset_A$. Attribute maps are essential for XSS analysis, because the executable part of most payloads resides in an attribute (for example, the `onerror` handler of an injected `img` element) rather than in the element's children or text.

Example Let's say we have a small DOM as listed below, where `cat` is the root element, and three children are present:

```

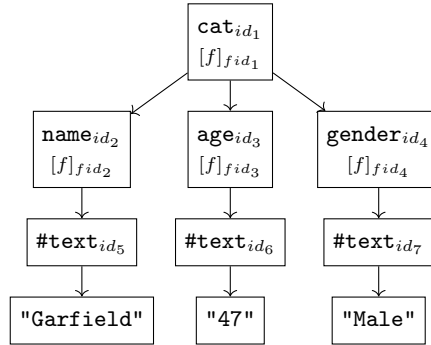
1 <cat>
2   <name>Garfield</name>
3   <age>47</age>
4   <gender>Male</gender>
5 </cat>

```

In the formal model, this structure is represented as a tree t of the form $s_{id}[A, f]_{fid}$. To construct the state, we begin with the root element name $s = \text{"cat"}$ and assign it a unique identifier id_1 . The element contains a completed forest $[f]_{fid_1}$, where fid_1 is the unique identifier for the child list. The forest f is composed using the forest composition operator $\otimes$. Since there are three children, the forest is built by applying the composition twice: $f = \langle t_1 \rangle_F \otimes \langle t_2 \rangle_F \otimes \langle t_3 \rangle_F$. Each child t_n is itself a tree containing a text node of the form $\#text_{id}[s]$. Ultimately, the representation in our DOM state space is as follows:

$$\begin{aligned}
&cat_{id_1} [\\
&\quad \langle name_{id_2} [\langle \#text_{id_5} ["Garfield"] \rangle_F]_{fid_2} \rangle_F \otimes \\
&\quad \langle age_{id_3} [\langle \#text_{id_6} ["47"] \rangle_F]_{fid_3} \rangle_F \otimes \\
&\quad \langle gender_{id_4} [\langle \#text_{id_7} ["Male"] \rangle_F]_{fid_4} \rangle_F \\
&]_{fid_1}
\end{aligned}$$

The above HTML code can be represented as a formal tree structure, highlighting the hierarchy and the unique identifiers (id and fid) as defined in the state space:



4.2 JavaScript as a State Space

While the DOM as a state space, as defined in the last section, provides a good structural environment for the webpage as such, XSS exploits are driven by JavaScript. JavaScript requires a different state space to model how payloads propagate from source to sink during XSS exploits. The JavaScript model will be defined as ρ , where ρ represents the *variable environment* [Plo04]. This approach is applicable for tracking data while propagating through the DOM.

Variable Environment The variable environment maps values to variable names. In DOM XSS context, primarily strings are important as strings can carry payloads for potential XSS exploits. The environment will be defined as a partial function:

$$\rho : \chi \rightharpoonup Val$$

where χ is the set of variable identifiers, including standalone variables, list items, field items and storage elements:

$$\chi ::= x \mid x[i] \mid x.f \mid \tau.\chi$$

Consequently, τ represents the specific type of storage $\tau \in \{ \text{url}, \text{cookie}, \text{localStorage}, \text{dom}, \text{sessionStorage}, \text{referrer} \}$. Taking the previous example, variable `name` can be assigned the value "Garfield" in the following way:

$$\text{let name} = \text{"Garfield"} \Leftrightarrow \rho(\text{"name"}) = \text{"Garfield"}$$

Program Transitions JavaScript operations are modeled as atomic program transitions that transform the JavaScript state space. Formally, atomic programs are elements of the set Π_0 , and each program defines a relation over states ρ .

We classify atomic programs into three classes, using labels as elements of the following defined sets:

$$\Lambda_{src} \stackrel{\text{def}}{=} \{ \text{url}, \text{cookie}, \text{localStorage}, \text{sessionStorage}, \text{dom}, \text{referrer} \}$$

$$\Lambda_{flow} \stackrel{\text{def}}{=} \{ \text{declare}, \text{lit}, \text{assign}, \text{concat}, \text{trim}, \text{sanitize}, \text{encode}, \text{decode} \}$$

$$\Lambda_{sink} \stackrel{\text{def}}{=} \{ \text{innerHTML}, \text{outerHTML}, \text{document.write}, \text{eval}, \text{setTimeout}, \text{setAttribute} \}$$

Each atomic program α^λ carries a label λ drawn from the corresponding label set, specifying its semantic role. The three classes are:

- **Source** (α_{src}^λ , where $\lambda_{src} \in \Lambda_{src}$): Source operations introduce external input into the JavaScript state. The label λ_{src} specifies the origin of the input, essentially reading out the environment variables as specified by τ . Formally:

$$\mathbf{m}(\alpha_{src}^\lambda) \subseteq \rho \times \rho'$$

where ρ' extends ρ by binding a location $x \in \chi$ to a value obtained from the external environment indexed by λ_{src} .

- **Flow** (α_{flow}^λ , where $\lambda_{flow} \in \Lambda_{flow}$): Flow operations define intra-state data movement and transformation. The label λ_{flow} specifies the semantic effect of the operation. Each λ_{flow} induces a distinct state transition rule over the environment ρ :

$$\mathbf{m}(\alpha_{flow}^\lambda) \subseteq \rho \times \rho'$$

Typical instances include copying values between locations, accessing fields or indices, and string concatenation.

- **Sink** (α_{sink}^λ , where $\lambda_{sink} \in \Lambda_{sink}$): Sink operations take in values from the JavaScript state and interact with the DOM or execution environment in a way that may trigger code execution. The label λ_{sink} specifies the sink context. Formally:

$$\mathbf{m}(\alpha_{sink}^\lambda) \subseteq (g, \rho) \times (g', \rho)$$

where the operation may additionally transform the DOM grove g .

Program execution is defined as relational composition of transitions:

$$\mathbf{m}(\alpha_1; \alpha_2) = \mathbf{m}(\alpha_1) \circ \mathbf{m}(\alpha_2)$$

An execution trace $\alpha_1; \dots; \alpha_n$ leads to a DOM XSS exploit if there exists a sequence of transitions from a source-induced state to a state in which a malicious string is executed at a sink location [MMT08].

Example Let's take a small example, demonstrating how data can be retrieved (like a source) and mutated in a series of propagation steps within JavaScript code. In this example, the acquirement for a cat adoption certificate is illustrated:

```

1 function adoptCat() {
2   let catName = "Garfield";
3   let certificate = "Certificate for " + catName;
4   return certificate;
5 }
6 let finalDoc = adoptCat();

```

Consider having completed line 3, i.e. `let certificate = "Certificate for " + catName;`, the function call to `adoptCat()` has already occurred, as well as the declaration of variable `catName`. The variable environment, at this moment, is defined in the following way:

$$\rho = \{ \text{"catName"} \mapsto \text{"Garfield"}, \text{"certificate"} \mapsto \text{"Certificate for Garfield"} \}$$

4.3 Combined State Model

To formalize a complete DOM XSS exploit, the DOM state model and the JavaScript state model both have to be reconciled into one model. We will unify these models into combined model σ : This model integrates the structural aspects of the webpage in a DOM along the interactive aspects of JavaScript code, allowing for reasoning about both domains. The combined state is defined as a tuple:

$$\sigma = (g, \rho)$$

where:

- $g \in G$ is the DOM grove, which is the heap of all tree structures
- $\rho : \chi \rightarrow Val$ is the variable environment, mapping script variables to values

Interaction between Models The unification of the DOM state space model and JavaScript state space model is realized through specific atomic programs that act as an interface. We will distinguish between two primary directions of flow: from the DOM to JavaScript, and from JavaScript to the DOM.

DOM-to-JS: These programs retrieve data from the grove g or the browser environment and store it in ρ . Using this approach, XSS sources can be characterized. For example, program α_{src} might read the content of a text node (using the node's id) or a location property, defined as:

$$\mathbf{m}(\text{read_source}(x, id)) = \{ ((g, \rho), (g, \rho[x \mapsto \text{lookup}(g, id)])) \}$$

JS-to-DOM: These programs take a value from ρ and use it to mutate g . This is where the actual DOM XSS occurs, since writing out from JavaScript to the DOM represents sink behavior. A sink execution like `innerHTML` can be modeled as a transition where a string in ρ is parsed into a new forest f and attached to a node in g . This results in a new DOM mutated result, namely g' :

$$\mathbf{m}(\text{write_sink}(id, x)) = \{ ((g, \rho), (g', \rho)) \}$$

where $g' = g[\text{parse}(\rho(x))/id]$ represents the grove g where the subtree at id is replaced by the parsed execution result of string variable x .

Flow operations propagate values within ρ without necessarily introducing external input or DOM effects.

Defining an Exploit A vulnerable execution is defined as the existence of a program sequence that transforms an initial state $\sigma = (g, \rho)$ into a state in which a malicious string has been propagated from a source location in ρ to a sink location in g . In set-theoretic notation, this is expressed in Propositional Dynamic Logic as:

$$\mathbf{m}_{\mathfrak{R}}(\alpha_{src}^{\lambda_{src}}) \circ (\mathbf{m}_{\mathfrak{R}}(\alpha_{flow}^{\lambda_{flow}}))^* \circ \mathbf{m}_{\mathfrak{R}}(\alpha_{sink}^{\lambda_{sink}}) \circ \text{Exploit}$$

As a satisfiability relation:

$$\sigma \models \langle \alpha_{src}^{\lambda_{src}}; (\alpha_{flow}^{\lambda_{flow}})^*; \alpha_{sink}^{\lambda_{sink}} \rangle \text{Exploit}$$

where Exploit is a state where in node in grove g contains the malicious string s_{mal} in execution context:

Example Consider a script that retrieves a string from the URL and injects it into a specific node: `let n = location.hash; node.innerHTML = n;`. Initially, the system is in state $\sigma_0 = (g_0, \rho_0)$. Grove g_0 is simply the structural representation of the target node's content, specifically a target element t_{id_1} defined as $\text{span}_{id_1}[\emptyset_A, \emptyset_F]_{fid_1}$. The variable environment ρ_0 is empty. When an attacker provides the payload "`<img src=x onerror=alert(1)>`", the source operation α_{src} transitions the system to σ_1 , where ρ is updated: $\rho(n) \mapsto \text{""}$. Note that at this stage, the DOM state g_0 remains secure as the malicious string only exists within the JavaScript environment. The utility of the DOM state space becomes clear during the sink operation α_{sink} . This operation does not simply replace a value: it invokes the DOM parsing to transform the string from ρ into a new formal tree structure. The transition $\mathbf{m}_{\mathfrak{R}}(\text{write_sink}(id_1, n))$ results in a new grove g_1 where the previously empty forest $[f]_{fid_1}$ is replaced by a forest containing the newly generated tree:

$$g_1 = g_0[fid_1 \mapsto \langle \text{img}_{id_2}[\{\text{onerror} \mapsto \text{"alert(1)"}\}, \emptyset_F]_{fid_2} \rangle_F]$$

The state $\sigma_0 \xrightarrow{\alpha_{src}} \sigma_1 \xrightarrow{\alpha_{sink}} \sigma_2$ satisfies the exploit proposition because the formal state space of the DOM now contains an executable node id_2 within the hierarchy of id_1 . The malicious value is no longer an inert string in ρ but resides in the attribute map $\{\text{onerror} \mapsto \text{"alert(1)"}\}$ of a parsed element, which is precisely the structural feature that makes it executable. By treating the DOM as a state space, we can precisely track how a string from the JavaScript environment is promoted to a structural element capable of causing code execution.

5 Formalization in Propositional Dynamic Logic

Given the conceptual framework as established in the previous chapter, the actual formalization in Propositional Dynamic Logic can now be constructed. Throughout this chapter, a formal definition of the combined state σ will initially be created, thereby consequently providing the basis for reasoning about Document Object Model and JavaScript transitions.

5.1 Formal Definition of the Combined State

5.1.1 Atomic Propositions

The combined state $\sigma = (g, \rho)$ is encoded into the Kripke Model $\mathfrak{K} = (K, R_{\mathfrak{K}}, \mathbf{m}_{\mathfrak{K}})$ by expressing each of its two components as families of atomic propositions in Φ_0 . We define two disjoint proposition families $\Phi_g, \Phi_\rho \subset \Phi_0$ such that $\Phi_0 = \Phi_g \cup \Phi_\rho$. For each atomic proposition $p \in \Phi_0$, the valuation assigns a set of states $\mathbf{m}_{\mathfrak{K}}(p) \subseteq K$, where $\sigma \in \mathbf{m}_{\mathfrak{K}}(p)$ if and only if $\sigma \models p$.

DOM Grove The DOM grove $g \in G$ is encoded via propositions in Φ_g that capture the security-relevant structural properties of nodes, in line with the local state space approach of the conceptual model. Let $\mathcal{ID}$ denote the universe of node identifiers and $\mathcal{A}$ the universe of attribute names, where we limit the set of attribute names as $\mathcal{A} \stackrel{\text{def}}{=} \{\text{onerror}, \text{onload}, \text{onclick}, \text{onmouseover}, \text{href}, \text{src}, \text{action}\}$. Moreover, Λ_{sink} is the set of sink labels and S the set of strings. We define $\text{nodes}(g) \stackrel{\text{def}}{=} \{id \in \mathcal{ID} \mid g \text{ contains a node with identifier } id\}$. The proposition set is then:

$$\begin{aligned} \Phi_g &\stackrel{\text{def}}{=} \{\text{node}_{id} \mid id \in \mathcal{ID}\} \\ &\cup \{\text{child}_{id_1, id_2} \mid id_1, id_2 \in \mathcal{ID}\} \\ &\cup \{\text{attr}_{id, a, s} \mid id \in \mathcal{ID}, a \in \mathcal{A}, s \in S\} \\ &\cup \{\text{sinknode}_{id, \lambda_{\text{sink}}, x} \mid id \in \mathcal{ID}, \lambda_{\text{sink}} \in \Lambda_{\text{sink}}, x \in \chi\} \end{aligned}$$

The valuation of each proposition set member is defined directly as:

$$\begin{aligned} \mathbf{m}_{\mathfrak{K}}(\text{node}_{id}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \exists s, A, f, fid : s_{id}[A, f]_{fid} \in g \vee \exists s : \# \text{text}_{id}[s] \in g\} \\ \mathbf{m}_{\mathfrak{K}}(\text{child}_{id_1, id_2}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \exists s, A, fid, f_1, f_2 : s_{id_1}[A, f_1 \otimes \langle t_{id_2} \rangle_F \otimes f_2]_{fid} \in g\} \\ \mathbf{m}_{\mathfrak{K}}(\text{attr}_{id, a, s}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \exists s', A, f, fid : s'_{id}[A, f]_{fid} \in g \wedge A(a) = s\} \\ \mathbf{m}_{\mathfrak{K}}(\text{sinknode}_{id, \lambda_{\text{sink}}, x}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \exists (g_0, \rho_0) \in K : ((g_0, \rho_0), (g, \rho)) \in R_{\alpha_{\text{sink}}^{\lambda_{\text{sink}}}(id, x)} \\ &\quad \wedge id \notin \text{nodes}(g_0) \wedge id \in \text{nodes}(g)\} \end{aligned}$$

where g_0 is the structural representation of the target node's content. Note that $\text{sinknode}_{id, \lambda_{\text{sink}}, x}$ encodes the sink labels $\lambda_{\text{sink}} \in \{\text{innerHTML}, \text{outerHTML}, \text{eval}, \text{document.write}\}$ from the conceptual model. A node satisfies $\text{sinknode}_{id, \lambda_{\text{sink}}, x}$ when it is in execution context corresponding to λ_{sink} and was produced by the sink that processed x .

Variable Environment ρ The variable environment $\rho : \chi \rightarrow Val$ is encoded as propositions in Φ_ρ , one independent set per variable $x \in \chi$. Each proposition makes a single yes/no statement about the current state of one particular variable. The full picture of ρ at any world σ is therefore constructed from the aggregation of all satisfied propositions across all $x \in \chi$, which is precisely how Kripke semantics encodes the state using atomic propositions. We define:

$$\begin{aligned} \Phi_\rho &\stackrel{\text{def}}{=} \{\text{bound}_x \mid x \in \chi\} \\ &\cup \{\text{val}_{x, s} \mid x \in \chi, s \in S\} \\ &\cup \{\text{tainted}_{x, \lambda_{\text{src}}} \mid x \in \chi, \lambda_{\text{src}} \in \Lambda_{\text{src}}\} \\ &\cup \{\text{match}_{x, \text{san}} \mid x \in \chi, \text{san} \in \mathbf{San}\} \end{aligned}$$

where $\Lambda_{src} = \{\text{url}, \text{cookie}, \text{localStorage}, \text{sessionStorage}, \text{dom}, \text{referrer}\}$ represents the source labels λ_{src} from the conceptual model. The valuations are:

$$\begin{aligned} \mathbf{m}_{\mathfrak{R}}(\text{bound}_x) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid x \in \text{dom}(\rho)\} \\ \mathbf{m}_{\mathfrak{R}}(\text{val}_{x,s}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \rho(x) = s\} \\ \mathbf{m}_{\mathfrak{R}}(\text{match}_{x,\text{san}}) &\stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \exists e \in \text{san} : \rho(x) \text{ is matched by } e\} \end{aligned}$$

San denotes the set of all sanitizer instances, where each $\text{san} \in \text{San}$ is a finite set of regular expressions $\text{san} = \{e_1, \dots, e_n\} \subset \mathcal{E}$, and $\mathcal{E}$ denotes the set of all regular expressions over S .

The taint valuation is defined separately. A variable is considered tainted if a source made it tainted, or if it received its value from another tainted variable. We write this as the smallest family of sets $\{T_{x,\lambda}\}$ that satisfies the four cases below, where "smallest" means a state counts as tainted only if one of the cases forces it to:

$$\begin{aligned} \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{x,\lambda}) &\stackrel{\text{def}}{=} T_{x,\lambda} \cdot \{ \sigma' \in K \mid \exists \sigma : \sigma \xrightarrow{\alpha_{src}^\lambda(x)} \sigma' \} \\ &\cup \{ \sigma' \in K \mid \exists \sigma, y : \sigma \xrightarrow{\alpha_{flow}^a(x,y)} \sigma', a \in \{\text{assign}, \text{trim}, \text{encode}, \text{decode}\}, \sigma \in T_{y,\lambda} \} \\ &\cup \{ \sigma' \in K \mid \exists \sigma, y_1, y_2 : \sigma \xrightarrow{\alpha_{flow}^{\text{concat}}(x,y_1,y_2)} \sigma', \sigma \in T_{y_1,\lambda} \vee \sigma \in T_{y_2,\lambda} \} \\ &\cup \{ \sigma' \in K \mid \exists \sigma, \beta : \sigma \xrightarrow{\beta} \sigma', \text{target}(\beta) \neq x, \sigma \in T_{x,\lambda} \} \end{aligned}$$

In the taint valuation, the four cases read as follows: a source operation makes x tainted; an **assign**, **trim**, **encode**, or **decode** copies the taint of y onto x ; a **concat** makes x tainted if either input was tainted; and the last case keeps existing taint when a step does not change x . The arrow $\sigma \xrightarrow{\beta} \sigma'$ means $(\sigma, \sigma') \in \mathbf{m}_{\mathfrak{R}}(\beta)$ for a program β , and $\text{target}(\beta)$ is the variable that β writes to: it is x for $\alpha_{src}^\lambda(x)$ and for any $\alpha_{flow}^a(x, \dots)$ with $a \in \{\text{declare}, \text{lit}, \text{assign}, \text{concat}, \text{trim}, \text{encode}, \text{decode}, \text{san-applied}\}$, and is undefined for **san-bypass**, tests, and sink programs.

5.1.2 Initial and Exploit States

Initial State The initial state $\sigma_0 \in K$ represents the system prior to any exploit-relevant execution. Its valuation is characterised by:

$$\begin{aligned} \sigma_0 &\notin \mathbf{m}_{\mathfrak{R}}(\text{bound}_x) && \forall x \in \chi \\ \sigma_0 &\in \mathbf{m}_{\mathfrak{R}}(\text{node}_{id}) && \forall id \in \text{nodes}(g_0) \\ \sigma_0 &\notin \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{x,\lambda_{src}}) && \forall x \in \chi, \lambda_{src} \in \Lambda_{src} \\ \sigma_0 &\notin \mathbf{m}_{\mathfrak{R}}(\text{sinknode}_{id,\lambda_{sink},x}) && \forall id \in \mathcal{ID}, \lambda_{sink} \in \Lambda_{sink}, x \in \chi \end{aligned}$$

Exploit State A state $\sigma \in K$ is an *exploit state* if a tainted value has propagated to a node in sink execution context. In set-theoretic notation:

$$\text{Exploit} \stackrel{\text{def}}{=} \bigcup_{\substack{x \in \chi, \lambda_{src} \in \Lambda_{src} \\ id \in \mathcal{ID}, \lambda_{sink} \in \Lambda_{sink}}} \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{x,\lambda_{src}}) \cap \mathbf{m}_{\mathfrak{R}}(\text{sinknode}_{id,\lambda_{sink},x})$$

5.2 Semantics of DOM and JavaScript Transitions

The atomic propositions have been established for our $\mathfrak{K}$, however, transitions need to be constructed in order to move in between these states. As defined in the conceptualization phase, there are three classes of atomic programs: source, flow and sink. This section formalizes that conceptualization into Propositional Dynamic Logic as *named transition relations* on the combined state space K . Every label in Λ_{src} , Λ_{flow} and Λ_{sink} is assigned separate *inference rules*. The exploit result is expressed as a composition of programs $\mathbf{m}_{\mathfrak{K}}(\alpha_1; \alpha_2)$.

5.2.1 Transition Relations as Named Relations

Each label λ induces a binary relation $R_\lambda \subseteq K \times K$ that coincides with the PDL valuation $\mathbf{m}_{\mathfrak{K}}(\alpha^\lambda) \subseteq K \times K$. Writing $\sigma \xrightarrow{\alpha^\lambda} \sigma'$ for $(\sigma, \sigma') \in R_\lambda$, the affected combined state $\sigma = (g, \rho)$ is as follows:

- **Source transitions** ($\lambda \in \Lambda_{src}$): only ρ is modified; g remains unmodified. Formally, if $(g, \rho) \xrightarrow{\alpha_{src}^\lambda} (g', \rho')$ then $g' = g$.
- **Flow transitions** ($\lambda \in \Lambda_{flow}$): only ρ is modified; g remains unmodified. Formally, if $(g, \rho) \xrightarrow{\alpha_{flow}^\lambda} (g', \rho')$ then $g' = g$.
- **Sink transitions** ($\lambda \in \Lambda_{sink}$): only g is modified; ρ remains unmodified. Formally, if $(g, \rho) \xrightarrow{\alpha_{sink}^\lambda} (g', \rho')$ then $\rho' = \rho$.

Because of these constraints, the DOM grove and the variable environment change independently. The constraints also justify the separation between multiple classes of propositions $\Phi_0 = \Phi_g \cup \Phi_\rho$. Source or flow transitions cannot invalidate any Φ_g proposition, while sink transitions cannot invalidate any Φ_ρ proposition.

5.2.2 Source Transition Rules

Source operations introduce external data into the variable environment by reading from a browser storage location indexed by a label $\lambda_{src} \in \Lambda_{src}$. All source locations are treated uniformly. That is, any of them may carry attacker-controlled data, regardless of whether the value originates from the URL, a cookie, or `localStorage`.

The semantics of a source transition depends on a partial *read function*:

$$\text{read} : \Lambda_{src} \rightarrow S$$

that retrieves the current value associated with the storage location indexed by λ_{src} . The function is partial because some locations may genuinely be undefined at execution time. `localStorage` or `sessionStorage`, for instance, may not contain the requested key, whereas others, such as `location.hash`, are always defined, though possibly equal to the empty string ε .

Rule S-READ handles the case where the storage location is defined:

$$\frac{\text{read}(\lambda_{src}) = s \quad x \notin \text{dom}(\rho)}{\langle g, \rho \rangle \xrightarrow{\alpha_{src}^{\lambda_{src}}(x)} \langle g, \rho[x \mapsto s] \rangle} \text{ S-READ}$$

Rule S-DEFAULT handles the case where the location is undefined; the variable is bound to the empty string so that subsequent flow operations are well-defined:

$$\frac{\text{read}(\lambda_{src}) = \perp}{\langle g, \rho \rangle \xrightarrow{\alpha_{src}^{\lambda_{src}}(x)} \langle g, \rho[x \mapsto \varepsilon] \rangle} \text{ S-DEFAULT}$$

Rule S-READ requires $x \notin \text{dom}(\rho)$, reflecting the standard semantics of a **let**-binding in JavaScript: a variable may not be re-declared in the same scope. Reassignment is handled instead by rule F-ASSIGN in the next subsection.

5.2.3 Flow Transition Rules

Flow operations propagate and transform values within the variable environment without introducing new external input or interacting with the DOM. A separate semantic definition is given for each label in $\Lambda_{\text{flow}} = \{\text{declare}, \text{lit}, \text{assign}, \text{concat}, \text{trim}, \text{sanitize}, \text{encode}, \text{decode}\}$.

Declaration The declare operation creates a new variable without value binding.

$$\frac{x \notin \text{dom}(\rho)}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{declare}}(x)} \langle g, \rho[x \mapsto \perp] \rangle} \text{ F-DECLARE}$$

Literal The lit operation declares a variable and immediately binds a literal value to it.

$$\frac{x \notin \text{dom}(\rho)}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{lit}}(x,s)} \langle g, \rho[x \mapsto s] \rangle} \text{ F-LIT}$$

Assignment The assign operation copies the value of an origin variable y into target variable x :

$$\frac{\rho(y) = s}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{assign}}(x,y)} \langle g, \rho[x \mapsto s] \rangle} \text{ F-ASSIGN}$$

Concatenation The concat operation appends two strings:

$$\frac{\rho(y_1) = s_1 \quad \rho(y_2) = s_2}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{concat}}(x,y_1,y_2)} \langle g, \rho[x \mapsto s_1 \cdot s_2] \rangle} \text{ F-CONCAT}$$

where $\cdot$ is the string concatenation operator.

Trim The trim operation removes a fixed prefix or suffix s_{fix} from the value of y , where **side** $\in \{\text{pre}, \text{suf}\}$ indicates which end is trimmed. Let $\text{trim}(\text{side}, s, s_{fix})$ denote the string obtained by removing one prefix-side occurrence of s_{fix} from s if **side** = **pre**, or one suffix-side occurrence if **side** = **suf**, and s otherwise:

$$\frac{\rho(y) = s \quad s' = \text{trim}(\text{side}, s, s_{fix})}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{trim}}(x,y,\text{side},s_{fix})} \langle g, \rho[x \mapsto s'] \rangle} \text{ F-TRIM}$$

Sanitization A sanitizer **san** $\in \mathbf{San}$ is a finite set of regular expressions $\{e_1, \dots, e_n\} \subset \mathcal{E}$ that models the patterns a developer intends to block. The sanitization program branches on whether the current value of x is matched by any expression in **san**, as determined by the proposition $\text{match}_{x,\text{san}}$.

Rule F-SAN-APPLIED handles the case where at least one pattern matches:

$$\frac{\rho(x) = s \quad \exists e \in \text{san} : e(s)}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{flow}}^{\text{san-applied}}(x,\text{san})} \langle g, \rho' \rangle} \text{ F-SAN-APPLIED}$$

Rule F-SAN-BYPASS handles the case where no pattern matches:

$$\frac{\rho(x) = s \quad \forall e \in \mathbf{san} : \neg e(s)}{\langle g, \rho \rangle \xrightarrow{\alpha_{flow}^{\mathbf{san-bypass}}(x)} \langle g, \rho \rangle} \text{ F-SAN-BYPASS}$$

The full sanitization program can be expressed as:

$$\alpha_{flow}^{\mathbf{sanitize}}(x, \mathbf{san}) \stackrel{\text{def}}{=} \mathbf{match}_{x, \mathbf{san}}? ; \alpha_{flow}^{\mathbf{san-applied}}(x, \mathbf{san}) \cup \neg \mathbf{match}_{x, \mathbf{san}}? ; \alpha_{flow}^{\mathbf{san-bypass}}(x)$$

Encoding and Decoding The encode and decode operations apply functions $\mathbf{enc}, \mathbf{dec} : S \rightarrow S$ corresponding to URL-encoding/decoding, HTML-entity encoding/decoding and Base64 encoding/decoding:

$$\frac{\rho(y) = s \quad \mathbf{enc}(s) = s'}{\langle g, \rho \rangle \xrightarrow{\alpha_{flow}^{\mathbf{encode}}(x, y)} \langle g, \rho[x \mapsto s'] \rangle} \text{ F-ENCODE} \quad \frac{\rho(y) = s \quad \mathbf{dec}(s) = s'}{\langle g, \rho \rangle \xrightarrow{\alpha_{flow}^{\mathbf{decode}}(x, y)} \langle g, \rho[x \mapsto s'] \rangle} \text{ F-DECODE}$$

Although $\mathbf{san}$ and $\mathbf{enc}$ both produce strings that are (in some sense) ‘safer’ than their inputs, they are *not* inverses of one another, and neither is an inverse of $\mathbf{dec}$. In particular, a sanitizer designed to strip angle brackets will not remove a URL-encoded angle bracket `%3C` from its input; applying $\mathbf{dec}$ *after* $\mathbf{san}$ can therefore restore the original dangerous character. This asymmetry is the basis for double-decode evasion techniques.

5.2.4 Sink Transition Rules

Sink operations read a value from the variable environment and use it to mutate the DOM grove, thereby crossing the JS-to-DOM boundary. The sink transition depends on the *parse function*:

$$\text{parse}_{\lambda_{\text{sink}}} : S \rightarrow F \cup G \cup (\mathcal{A} \times S)$$

that converts a string into a forest with attributes suitable for insertion into the grove. In case an error signal arises or no execution is performed, the forest will stay empty $f = \emptyset_F$. The parse function varies by sink label, reflecting the distinct browser behaviours as shown in the table below.

Table 1: Parse function behavior by sink label

Sink label	$\text{parse}_{\lambda}(s)$ behavior
<code>innerHTML</code>	Full HTML parse of s ; produces a forest f containing element nodes, text nodes, and event-handler attributes.
<code>outerHTML</code>	Like <code>innerHTML</code> but replaces the target node itself; the parse result is grafted one level higher in the grove.
<code>eval</code>	No DOM mutation (g is unchanged); s is treated as a new JavaScript source text.
<code>document.write</code>	Replaces the <i>entire</i> grove g with the parse result of s , rather than a single subtree.
<code>setAttribute</code>	Modifies the attribute map A of an existing node; the forest structure of g is unchanged.

Standard sink rule For the labels `innerHTML`, `outerHTML`, and `setAttribute`, the transition has the following general form:

$$\frac{\rho(x) = s \quad \text{parse}_{\lambda_{\text{sink}}}(s) = f \quad id \in \text{nodes}(g)}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{sink}}^{\lambda_{\text{sink}}}(id, x)} \langle g[fid_{id} \mapsto f], \rho \rangle} \text{ K-SINK}$$

Here $g[fid_{id} \mapsto f]$ denotes the grove obtained from g by replacing the child-forest stored at forest identifier fid_{id} , i.e. the children of node id , with the newly parsed HTML forest f . The variable environment ρ remains unchanged since its variables are only used and not altered.

document.write Because `document.write` replaces the entire grove rather than a single subtree, it requires a separate rule:

$$\frac{\rho(x) = s \quad \text{parse}_{\text{document.write}}(s) = g'}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{sink}}^{\text{document.write}}(x)} \langle g', \rho \rangle} \text{K-WRITE}$$

This rule does not require a target node identifier, because the entire grove is replaced unconditionally.

eval The `eval` sink does not mutate g directly. Instead, the string $\rho(x)$ is treated as a new JavaScript program text and executed in the current environment. Formally expressed:

$$\frac{\rho(x) = s \quad \text{js_parse}(s) = \beta \quad \langle g, \rho \rangle \xrightarrow{\beta} \langle g', \rho' \rangle}{\langle g, \rho \rangle \xrightarrow{\alpha_{\text{sink}}^{\text{eval}}(x)} \langle g', \rho' \rangle} \text{K-EVAL}$$

The attacker-controlled string s is itself a program that may contain further source, flow, and sink operations. In PDL notation, the reachability condition for an `eval`-driven exploit can be defined as the satisfiability relation:

$$\sigma \models \langle \alpha_{\text{src}}^{\lambda_{\text{src}}}; (\alpha_{\text{flow}}^{\lambda_{\text{flow}}})^*; \alpha_{\text{sink}}^{\text{eval}} \rangle \langle \alpha_{\text{src}}^{\lambda'_{\text{src}}}; (\alpha_{\text{flow}}^{\lambda'_{\text{flow}}})^*; \alpha_{\text{sink}}^{\lambda'_{\text{sink}}} \rangle \text{Exploit}$$

5.2.5 Taint-Propagation Constraints

The transition rules above specify what happens to ρ and g at each step, but they do not by themselves determine whether a value is attacker-controlled. To answer that question, taint propagation is defined as a set of constraints on the atomic proposition $\text{tainted}_{x, \lambda_{\text{src}}}$, specifying which propositions must hold in the next state σ' given which propositions held in the current state σ , for each transition rule.

Source A source transition introduces taint on the bound variable:

$$\sigma \xrightarrow{\alpha_{\text{src}}^{\lambda_{\text{src}}}(x)} \sigma' \implies \sigma' \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{x, \lambda_{\text{src}}})$$

Declaration A bare declaration binds the variable to $\perp$ and introduces no value, so it introduces no taint:

$$\sigma \xrightarrow{\alpha_{\text{flow}}^{\text{declare}}(x)} \sigma' \implies \forall \lambda \in \Lambda_{\text{src}} : \sigma' \notin \mathbf{m}_{\mathcal{R}}(\text{tainted}_{x, \lambda})$$

Literal A literal-operation creates a new variable and binds a fixed value to it. Since its value doesn't come from user input (as it would with source operations), the value is not tainted:

$$\sigma \xrightarrow{\alpha_{\text{flow}}^{\text{lit}}(x, s)} \sigma' \implies \forall \lambda \in \Lambda_{\text{src}} : \sigma' \notin \mathbf{m}_{\mathcal{R}}(\text{tainted}_{x, \lambda})$$

Assignment Taint propagates from the source variable to the target. Any prior taint on x is overwritten:

$$\sigma \xrightarrow{\alpha_{\text{flow}}^{\text{assign}}(x, y)} \sigma' \implies \forall \lambda \in \Lambda_{\text{src}} : \sigma' \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{x, \lambda}) \iff \sigma \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{y, \lambda})$$

Concatenation Taint is preserved through concatenation: if either operand is tainted, the result is tainted. Appending a tainted string to a different string does not neutralize the taint:

$$\begin{aligned} \sigma \xrightarrow{\alpha_{flow}^{concat}(x,y_1,y_2)} \sigma' &\implies \forall \lambda \in \Lambda_{src} : \sigma' \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda}) \\ &\iff \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{y_1,\lambda}) \vee \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{y_2,\lambda}) \end{aligned}$$

Trim Trim preserves taint, because removing a fixed prefix or suffix does not render the remaining attacker-controlled content safe:

$$\sigma \xrightarrow{\alpha_{flow}^{trim}(x,y,\mathbf{side},\mathbf{suffix})} \sigma' \implies \forall \lambda \in \Lambda_{src} : \sigma' \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda}) \iff \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{y,\lambda})$$

Sanitization The taint outcome of a sanitization step depends on which branch was taken. If the match branch is true, taint is removed:

$$\sigma \xrightarrow{\alpha_{flow}^{san-applied}(x,\mathbf{san})} \sigma' \implies \forall \lambda \in \Lambda_{src} : \sigma' \notin \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda})$$

If the bypass branch is true, taint is fully preserved:

$$\sigma \xrightarrow{\alpha_{flow}^{san-bypass}(x)} \sigma' \implies \forall \lambda \in \Lambda_{src} : \sigma' \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda}) \iff \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda})$$

In both cases, sanitization affects only the variable x passed to the program. Any other variable that carries the same tainted value independently is unaffected, and can still be passed to a sink separately.

Encoding and Decoding Both encode and decode transitions preserve taint, because neither operation renders a string completely safe. In particular, applying **dec** after **san** may restore dangerous characters that the sanitizer had removed, which is the basis of double-decode evasion techniques discussed. Formally, encoding and decoding can be expressed as:

$$\begin{aligned} \sigma \xrightarrow{\alpha_{flow}^{encode}(x,y)} \sigma' &\implies \forall \lambda \in \Lambda_{src} : \sigma' \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda}) \iff \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{y,\lambda}) \\ \sigma \xrightarrow{\alpha_{flow}^{decode}(x,y)} \sigma' &\implies \forall \lambda \in \Lambda_{src} : \sigma' \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x,\lambda}) \iff \sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{y,\lambda}) \end{aligned}$$

Sink A sink transition does not affect ρ , so no variable gains or loses taint. However, the proposition $\mathbf{sinknode}_{id,\lambda_{sink},x}$ becomes satisfied in σ' for every node id newly introduced into g by the transition. In case $\mathbf{sinknode}_{id,\lambda_{sink},x}$ and $\mathbf{tainted}$ propositions are both true, the exploit is successful.

6 Validation

To demonstrate the PDL formalization, examples are used to trace a complete DOM XSS exploit through every layer of the model: the combined state, the transition rules, the taint-propagation constraints, and the final exploit proposition. The first example involves a concatenation step and a sanitization attempt that is rendered ineffective by the way in which operations are applied. The second example shows an exploit where multiple execution paths could be taken, thereby providing a non-linear exploit. The third example shows that not all exploits could be expressed in this model, highlighting this model's limits.

6.1 Example One: Sanitization Wrongfully Applied

In this first example, a DOM XSS exploit is expressed based on a real-world example. A few string-manipulation operations are performed first. Then, sanitization gets applied on the wrong variable. Finally, the malicious string s_{mal} reaches the sink via `innerHTML`.

The Vulnerable Script

Consider the following script:

```

1 let raw    = location.hash.substring(1);
2 let prefix = "Input: ";
3 let msg    = prefix + raw;
4 let safe   = sanitize(raw);
5 document.getElementById("output").innerHTML = msg;

```

The developer intends to display user-supplied input inside a `div` element. A sanitizer is applied to `raw` on line 4, but the result `safe` is never used: the unsanitized concatenation `msg` is passed to `innerHTML` on line 5. This is a common programming mistake and constitutes a real DOM XSS vulnerability.

Initial State σ_0

The grove g_0 contains a single `div` element with no children, identified by id_1 :

$$g_0 = \text{div}_{id_1}[\emptyset_A, \emptyset_F]_{fid_1}$$

The variable environment is empty:

$$\rho_0 = \emptyset$$

The propositions satisfied in $\sigma_0 = (g_0, \rho_0)$ are:

$$\begin{aligned}
\sigma_0 &\in \mathbf{m}_{\mathcal{R}}(\text{node}_{id_1}) \\
\sigma_0 &\notin \mathbf{m}_{\mathcal{R}}(\text{bound}_x) & \forall x \in \chi \\
\sigma_0 &\notin \mathbf{m}_{\mathcal{R}}(\text{tainted}_{x,\lambda}) & \forall x \in \chi, \lambda \in \Lambda_{src}
\end{aligned}$$

State σ_1 : Line 1 — Source Operation

Line 1 reads `location.hash` and binds the result to `raw`. This is modelled by rule S-READ with $\lambda_{src} = \text{url}$, where s_{mal} is the attacker-supplied payload (e.g. `<img src=x onerror=alert(1)>`):

$$\frac{\text{read}(\text{url}) = s_{mal} \quad \text{raw} \notin \text{dom}(\rho_0)}{\langle g_0, \rho_0 \rangle \xrightarrow{\alpha_{src}^{\text{url}}(\text{raw})} \langle g_0, \rho_1 \rangle} \text{S-READ}$$

where $\rho_1 = \rho_0[\text{raw} \mapsto s_{mal}]$. The grove is unchanged: $g_1 = g_0$. By the source taint constraint, the bound variable immediately acquires taint:

$$\sigma_1 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{raw}, \text{url}})$$

State σ_2 : Line 2 — Flow Operation (Literal)

Line 2 binds the string literal "Input: " to **prefix**. A literal has a fixed value originating from the program text itself, not from any external location, so it is modelled as a flow operation introducing a fresh untainted binding:

$$\frac{\text{prefix} \notin \text{dom}(\rho_1)}{\langle g_0, \rho_1 \rangle \xrightarrow{\alpha_{flow}^{lit}(\text{prefix}, \text{"Input: "})} \langle g_0, \rho_2 \rangle} \text{F-LIT}$$

where $\rho_2 = \rho_1[\text{prefix} \mapsto \text{"Input: "}]$. The grove is unchanged: $g_2 = g_0$. Since a literal carries no external input, **prefix** acquires no taint:

$$\sigma_2 \notin \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{prefix}, \lambda}) \quad \forall \lambda \in \Lambda_{src}$$

State σ_3 : Line 3 — Flow Operation (Concatenation)

Line 3 concatenates **prefix** and **raw** into **msg**, modelled by rule F-CONCAT:

$$\frac{\rho_2(\text{prefix}) = \text{"Input: "} \quad \rho_2(\text{raw}) = s_{mal}}{\langle g_0, \rho_2 \rangle \xrightarrow{\alpha_{flow}^{concat}(\text{msg}, \text{prefix}, \text{raw})} \langle g_0, \rho_3 \rangle} \text{F-CONCAT}$$

where $\rho_3 = \rho_2[\text{msg} \mapsto \text{"Input: "} \cdot s_{mal}]$. By the concatenation taint constraint, **msg** inherits the taint of both operands:

$$\sigma_3 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{msg}, \lambda}) \iff \sigma_2 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{prefix}, \lambda}) \vee \sigma_2 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{raw}, \lambda}) \quad \forall \lambda \in \Lambda_{src}$$

Since $\sigma_2 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{raw}, \text{url}})$, it follows in particular that:

$$\sigma_3 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{msg}, \text{url}})$$

Prepending the fixed string "Input: " does not neutralise the taint, because the attacker-controlled string is preserved in **msg**.

State σ_4 : Line 4 — Flow Operation (Sanitization)

Line 4 applies the sanitizer to **raw**, with the result bound to **safe**. We assume the sanitizer **san**₁ contains patterns that match s_{mal} , so the proposition **match**_{**raw**, **san**₁} holds in σ_3 . The F-SAN-APPLIED branch is therefore taken:

$$\frac{\rho_3(\text{raw}) = s_{mal} \quad \exists e \in \text{san}_1 : e(s_{mal})}{\langle g_0, \rho_3 \rangle \xrightarrow{\alpha_{flow}^{san-applied}(\text{raw}, \text{san}_1)} \langle g_0, \rho_4 \rangle} \text{F-SAN-APPLIED}$$

where $\rho_4 = \rho_3$, since sanitization does not mutate any variable binding. By the sanitization taint constraint for F-SAN-APPLIED:

$$\sigma_4 \notin \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{raw}, \lambda}) \quad \forall \lambda \in \Lambda_{src}$$

Crucially, the sanitizer was applied to **raw**, not to **msg**. The taint constraint affects only the variable passed to the program, so the taint of **msg** is entirely unaffected:

$$\sigma_4 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{msg}, \text{url}})$$

The variable **safe** from the original script is no longer modelled as a separate binding, since the new sanitization program operates directly on **raw** without producing a new output variable. The sanitization is therefore entirely ineffective with respect to the exploit: the tainted variable **msg** remains unaffected and will be passed to the sink.

State σ_5 : Line 5 — Sink Operation

Line 5 writes `msg` to the `innerHTML` of the target node, modelled by rule K-SINK:

$$\frac{\rho_4(\text{msg}) = \text{"Input: "} \cdot s_{mal} \quad \text{parse}_{\text{innerHTML}}(\text{"Input: "} \cdot s_{mal}) = f \quad id_1 \in \text{nodes}(g_0)}{\langle g_0, \rho_4 \rangle \xrightarrow{\alpha_{\text{sink}}^{\text{innerHTML}}(id_1, \text{msg})} \langle g_5, \rho_4 \rangle} \text{K-SINK}$$

The parse function interprets `"Input: " · smal` as HTML, producing a forest f containing a text node for the fixed prefix and an `img` element whose attribute map carries the executable `onerror` handler for the payload:

$$f = \langle \# \text{text}_{id_2}[\text{"Input: "}]_F \otimes \langle \text{img}_{id_3}[\{\text{onerror} \mapsto \text{"alert(1)"}\}, \emptyset_F]_{fid_3} \rangle_F$$

The resulting grove is:

$$g_5 = g_0[fid_1 \mapsto f]$$

Since the sink transition introduces node id_3 into the grove, the `sinknode` proposition becomes satisfied by the definition of $\mathbf{m}_{\mathcal{R}}(\text{sinknode}_{id, \lambda_{\text{sink}}})$: the node id_3 was absent from g_0 and is present in g_5 as a result of the `innerHTML` transition. Furthermore, ρ is unchanged by the sink transition, so the taint of `msg` persists:

$$\begin{aligned} \sigma_5 &\in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{msg}, \text{url}}) \\ \sigma_5 &\in \mathbf{m}_{\mathcal{R}}(\text{sinknode}_{id_3, \text{innerHTML}, \text{msg}}) \end{aligned}$$

Exploit State

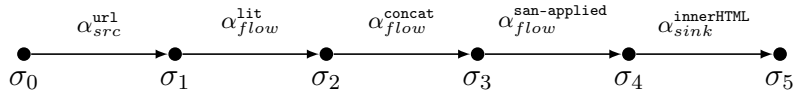
Since both propositions are simultaneously satisfied in σ_5 :

$$\sigma_5 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{msg}, \text{url}}) \cap \mathbf{m}_{\mathcal{R}}(\text{sinknode}_{id_3, \text{innerHTML}, \text{msg}}) \subseteq \text{Exploit}$$

The system has reached an exploit state. In PDL notation, the entire execution trace is expressed as:

$$\sigma_0 \models \left\langle \alpha_{\text{src}}^{\text{url}}(\text{raw}); \alpha_{\text{flow}}^{\text{lit}}(\text{prefix}, \text{"Input: "}); \alpha_{\text{flow}}^{\text{concat}}(\text{msg}, \text{prefix}, \text{raw}); \right. \\ \left. \alpha_{\text{flow}}^{\text{san-applied}}(\text{raw}, \text{san}_1); \alpha_{\text{sink}}^{\text{innerHTML}}(id_1, \text{msg}) \right\rangle \text{Exploit}$$

The sanitize step is present in the program sequence yet does not prevent the exploit. The PDL formula is satisfiable because the sanitizer was applied to `raw`, while it is `msg` that reaches the sink: `taintedraw, url` does not hold in σ_4 , but `taintedmsg, url` does hold simultaneously and it is `msg` that is passed to K-SINK.



	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5
<code>tainted_{raw, url}</code>	·	✓	✓	✓	·	·
<code>tainted_{msg, url}</code>	·	·	·	✓	✓	✓
<code>sinknode_{id₃, innerHTML, msg}</code>	·	·	·	·	·	✓
<code>Exploit</code>	·	·	·	·	·	✓

6.2 Example Two: Multiple pathways

Multiple exploit pathways are expressible in the formalization, which is demonstrated in this example using an if-statement. Also, the double decoding technique is often used by developers as a way to strengthen the webpage's security. However, this example shows that exploits are still possible if constructed precisely.

The Vulnerable Script

Consider the following script:

```

1  let raw = location.hash.substring(1);
2  let msg;
3  if (raw.substring(0, 5) == "user:") {
4      msg = raw.substring(5);
5  } else {
6      msg = raw;
7  }
8  let decoded = decodeURIComponent(msg);
9  let safe    = sanitize(decoded);
10 let output  = decodeURIComponent(safe);
11 document.getElementById("output").innerHTML = output;

```

The developer retrieves a URL fragment and conditionally strips a "user:" prefix. The value is then decoded once, passed through a sanitizer intended to block dangerous HTML, and decoded a second time before being written to `innerHTML`. The developer's assumption is that the sanitizer inspects the final form of the string. However, the sanitizer is applied between two decode operations. An attacker who submits a double-encoded payload such as `user:%253Cimg%2520src=x%2520onerror=alert(1)%253E` can exploit this ordering: the first decode produces a string, encoded once, namely `%3Cimg%20src=x%20onerror=alert(1)%3E`, which the sanitizer does not recognise as dangerous HTML and therefore passes. The second decode then restores the full dangerous string. This is a classical double-decode evasion pattern, and it is reachable via both branches of the conditional.

Initial State σ_0

The grove g_0 contains a single `div` element with no children or attributes, identified by id_1 :

$$g_0 = \text{div}_{id_1}[\emptyset_A, \emptyset_F]_{fid_1}$$

The variable environment is empty:

$$\rho_0 = \emptyset$$

The propositions satisfied in $\sigma_0 = (g_0, \rho_0)$ are:

$$\begin{array}{ll}
 \sigma_0 \in \mathbf{m}_{\mathfrak{R}}(\text{node}_{id_1}) & \\
 \sigma_0 \notin \mathbf{m}_{\mathfrak{R}}(\text{bound}_x) & \forall x \in \chi \\
 \sigma_0 \notin \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{x,\lambda}) & \forall x \in \chi, \lambda \in \Lambda_{src}
 \end{array}$$

State σ_1 : Line 1 — Source Operation

Line 1 reads `location.hash` and binds the result to `raw`. The attacker provides the double encoded payload `user:%253Cimg%2520src=x%2520onerror=alert(1)%253E`, where `%25` is percent-encoding of the percent sign `%`, so the outermost encoding layer hides the inner one. This is

modeled by rule S-READ with $\lambda_{src} = \text{url}$:

$$\frac{\text{read}(\text{url}) = s_{mal} \quad \text{raw} \notin \text{dom}(\rho_0)}{\langle g_0, \rho_0 \rangle \xrightarrow{\alpha_{src}^{\text{url}}(\text{raw})} \langle g_0, \rho_1 \rangle} \text{S-READ}$$

where $s_{mal} = \text{user}:\%253\text{Cimg}\%2520\text{src}=\text{x}\%2520\text{onerror}=\text{alert}(1)\%253\text{E}$ and $\rho_1 = \rho_0[\text{raw} \mapsto s_{mal}]$. The grove is unchanged: $g_1 = g_0$. By the source taint constraint:

$$\sigma_1 \in \mathbf{m}_{\mathcal{R}}(\mathbf{tainted}_{\text{raw}, \text{url}})$$

State σ_2 : Line 2 — Flow Operation (Declaration)

Line 2, `let msg;`, declares `msg` without binding a value. It is modelled by rule F-DECLARE, which binds the variable to $\perp$:

$$\frac{\text{msg} \notin \text{dom}(\rho_1)}{\langle g_0, \rho_1 \rangle \xrightarrow{\alpha_{flow}^{\text{declare}}(\text{msg})} \langle g_0, \rho_2 \rangle} \text{F-DECLARE}$$

where $\rho_2 = \rho_1[\text{msg} \mapsto \perp]$. The grove is unchanged: $g_2 = g_0$. Since a bare declaration introduces no value, `msg` acquires no taint:

$$\sigma_2 \notin \mathbf{m}_{\mathcal{R}}(\mathbf{tainted}_{\text{msg}, \lambda}) \quad \forall \lambda \in \Lambda_{src}$$

The value of `msg` is assigned subsequently, inside the branch taken in σ_3 , which overwrites the $\perp$ binding.

State σ_3 : Lines 3–6 — Flow Operation (Branching)

Lines 3–6 showcase an if-else branch on whether the first five characters of `raw` equal `"user:"`. In PDL, this is expressed using the choice operator, differentiating between conditions on the current value of `raw`:

$$\text{prefix?} ; \alpha_{flow}^{\text{trim}}(\text{msg}, \text{raw}, \text{pre}, \text{"user:"}) \cup \neg \text{prefix?} ; \alpha_{flow}^{\text{assign}}(\text{msg}, \text{raw})$$

where `prefix` is the proposition $\text{val}_{\text{raw}, \text{"user:"}.s}$ for some $s \in S$, i.e. `raw` begins with `"user:"`. Since the attacker's payload satisfies this condition, the if-branch is taken and rule F-TRIM applies:

$$\frac{\rho_2(\text{raw}) = s_{mal} \quad s_{enc} = \text{trim}(\text{pre}, s_{mal}, \text{"user:"})}{\langle g_0, \rho_2 \rangle \xrightarrow{\alpha_{flow}^{\text{trim}}(\text{msg}, \text{raw}, \text{pre}, \text{"user:"})} \langle g_0, \rho_3 \rangle} \text{F-TRIM}$$

where $s_{enc} = \%253\text{Cimg}\%2520\text{src}=\text{x}\%2520\text{onerror}=\text{alert}(1)\%253\text{E}$ and $\rho_3 = \rho_2[\text{msg} \mapsto s_{enc}]$. By the trim taint constraint, taint is preserved:

$$\sigma_3 \in \mathbf{m}_{\mathcal{R}}(\mathbf{tainted}_{\text{msg}, \text{url}})$$

Had the else-branch been taken instead, `msg` would have been assigned the full `raw` value directly via F-ASSIGN, so $\mathbf{tainted}_{\text{msg}, \text{url}}$ would still hold. The exploit is therefore reachable via both branches.

State σ_4 : Line 8 — Flow Operation (First Decode)

Line 8 applies `decodeURIComponent` to `msg` and binds the result to `decoded`, modelled by rule F-DECODE:

$$\frac{\rho_3(\text{msg}) = s_{enc} \quad \text{dec}(s_{enc}) = s_{enc1}}{\langle g_0, \rho_3 \rangle \xrightarrow{\alpha_{flow}^{\text{decode}}(\text{decoded}, \text{msg})} \langle g_0, \rho_4 \rangle} \text{F-DECODE}$$

where $s_{enc1} = \%3Cimg\%20src=x\%20onerror=alert(1)\%3E$ and $\rho_4 = \rho_3[\text{decoded} \mapsto s_{enc1}]$. The first decode removes the outer layer of percent-encoding, transforming $\%25$ back into $\%$, so the dangerous characters $<$ and $>$ remain encoded as $\%3C$ and $\%3E$. The string is not yet dangerous in its literal form. By the decode taint constraint, taint is preserved:

$$\sigma_4 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{decoded}, \text{url}})$$

State σ_5 : Line 9 — Flow Operation (Sanitization Bypass)

Line 9 applies the sanitizer to `decoded`, with the result subsequently bound to `safe`. The sanitizer `san1` assumes to contain patterns matching raw HTML strings such as `<img` and `onerror=`. However, the current value of `decoded` is the string s_{enc1} which has only been decoded once. Therefore, it does not contain literal angle brackets or the string `onerror=` in unencoded form. No pattern in `san1` matches, so the proposition `matchdecoded, san1` does not hold in σ_4 , and the F-SAN-BYPASS branch is taken:

$$\frac{\rho_4(\text{decoded}) = s_{enc1} \quad \forall e \in \text{san}_1 : \neg e(s_{enc1})}{\langle g_0, \rho_4 \rangle \xrightarrow[\alpha_{flow}^{\text{san-bypass}(\text{decoded})}]{} \langle g_0, \rho_4 \rangle} \text{F-SAN-BYPASS}$$

Since F-SAN-BYPASS leaves ρ unchanged, the binding of `safe` is modelled as a subsequent F-ASSIGN step:

$$\frac{\rho_4(\text{decoded}) = s_{enc1}}{\langle g_0, \rho_4 \rangle \xrightarrow[\alpha_{flow}^{\text{assign}(\text{safe}, \text{decoded})}]{} \langle g_0, \rho_5 \rangle} \text{F-ASSIGN}$$

where $\rho_5 = \rho_4[\text{safe} \mapsto s_{enc1}]$. By the bypass and assignment taint constraints respectively:

$$\sigma_5 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{decoded}, \text{url}}) \quad \wedge \quad \sigma_5 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{safe}, \text{url}})$$

This is the crucial step that distinguishes this example from Example 1. Here the sanitizer is not applied to the wrong variable: it inspects the correct variable but genuinely fails to detect the threat, because the payload is still in encoded form at the point of inspection. The bypass is not a programming error in the application of sanitization, but an evasion arising from the asymmetry between `san` and `dec`: the sanitizer was designed to match `<img`, but the value it sees is `\%3Cimg`.

State σ_6 : Line 10 — Flow Operation (Second Decode)

Line 10 applies `decodeURIComponent` a second time to `safe` and binds the result to `output`, modelled by a second application of rule F-DECODE:

$$\frac{\rho_5(\text{safe}) = s_{enc1} \quad \text{dec}(s_{enc1}) = s'_{mal}}{\langle g_0, \rho_5 \rangle \xrightarrow[\alpha_{flow}^{\text{decode}(\text{output}, \text{safe})}]{} \langle g_0, \rho_6 \rangle} \text{F-DECODE}$$

where $s'_{mal} = \text{}$ and $\rho_6 = \rho_5[\text{output} \mapsto s'_{mal}]$. The second decode fully restores the dangerous HTML string. Both encoding layers have now been fully decoded, and the sanitizer's inspection - which occurred between the two decodes - was therefore directed at an intermediate form that did not reveal the malicious content. By the decode taint constraint, taint is preserved:

$$\sigma_6 \in \mathbf{m}_{\mathcal{R}}(\text{tainted}_{\text{output}, \text{url}})$$

State σ_7 : Line 11 — Sink Operation

Line 11 writes `output` to the `innerHTML` of the target node, modelled by rule K-SINK:

$$\frac{\rho_6(\text{output}) = s'_{mal} \quad \text{parse}_{\text{innerHTML}}(s'_{mal}) = f \quad id_1 \in \text{nodes}(g_0)}{\langle g_0, \rho_6 \rangle \xrightarrow[\alpha_{sink}^{\text{innerHTML}(id_1, \text{output})}]{} \langle g_7, \rho_6 \rangle} \text{K-SINK}$$

The parse function interprets s'_{mal} as HTML, producing a forest f containing an `img` element with an executable `onerror` attribute:

$$f = \langle \text{img}_{id_2}[\{\text{onerror} \mapsto \text{"alert(1)"}\}, \emptyset_F]_{fid_2} \rangle_F$$

where `attr(onerror, "alert(1)")` is attached to `imgid2`. The resulting grove is:

$$g_7 = g_0[fid_1 \mapsto f]$$

Since the sink transition introduces node id_2 into the grove, the `sinknode` proposition becomes satisfied by the definition of $\mathbf{m}_{\mathfrak{R}}(\text{sinknode}_{id, \lambda_{sink}})$: the node id_2 was absent from g_0 and is present in g_7 as a result of the `innerHTML` transition. Furthermore, ρ is unchanged by the sink transition, so the taint of `output` persists:

$$\begin{aligned} \sigma_7 &\in \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{\text{output}, \text{url}}) \\ \sigma_7 &\in \mathbf{m}_{\mathfrak{R}}(\text{sinknode}_{id_2, \text{innerHTML}, \text{output}}) \end{aligned}$$

Exploit State

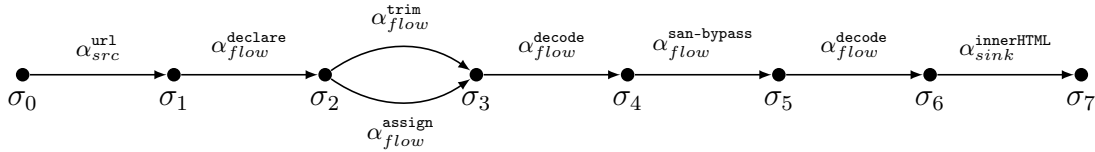
Since both propositions are simultaneously satisfied in σ_7 :

$$\sigma_7 \in \mathbf{m}_{\mathfrak{R}}(\text{tainted}_{\text{output}, \text{url}}) \cap \mathbf{m}_{\mathfrak{R}}(\text{sinknode}_{id_2, \text{innerHTML}, \text{output}}) \subseteq \text{Exploit}$$

The system has reached an exploit state. In PDL notation, the full execution trace is expressed as:

$$\begin{aligned} \sigma_0 \models & \left\langle \alpha_{src}^{\text{url}}(\text{raw}) ; (\text{prefix?} ; \alpha_{flow}^{\text{trim}}(\text{msg}, \text{raw}, \text{pre}, \text{"user:"}) \cup \neg \text{prefix?} ; \alpha_{flow}^{\text{assign}}(\text{msg}, \text{raw})) ; \right. \\ & \alpha_{flow}^{\text{decode}}(\text{decoded}, \text{msg}) ; \alpha_{flow}^{\text{san-bypass}}(\text{decoded}) ; \alpha_{flow}^{\text{assign}}(\text{safe}, \text{decoded}) ; \\ & \left. \alpha_{flow}^{\text{decode}}(\text{output}, \text{safe}) ; \alpha_{sink}^{\text{innerHTML}}(id_1, \text{output}) \right\rangle \text{Exploit} \end{aligned}$$

The $\cup$ operator makes explicit that the exploit is reachable via both branches of the if-else: regardless of whether the trim branch or the assignment branch is taken, `taintedmsg, url` holds in σ_3 and all subsequent steps preserve that taint through to the sink. The double-decode evasion is visible directly in the trace as two $\alpha_{flow}^{\text{decode}}$ steps with the sanitizer application in between them as $\alpha_{flow}^{\text{san-bypass}}$, which formally captures why the sanitizer provides no security guarantee when a second decode follows: the value it inspected was not the value that reached the sink.



	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7
<code>tainted_{raw, url}</code>	.	✓	✓	✓	✓	✓	✓	✓
<code>tainted_{msg, url}</code>	.	.	.	✓	✓	✓	✓	✓
<code>tainted_{decoded, url}</code>	.	.	.	.	✓	✓	✓	✓
<code>tainted_{safe, url}</code>	.	.	.	.	.	✓	✓	✓
<code>tainted_{output, url}</code>	.	.	.	.	.	.	✓	✓
<code>sinknode_{id₂, innerHTML, output}</code>	.	.	.	.	.	.	.	✓
Exploit	.	.	.	.	.	.	.	✓

6.3 Example Three: Formalization Limits

Although the formalization provides a quite comprehensive way of encapsulating DOM XSS exploits, it is not all-encompassing. Some variations of DOM XSS exploits cannot be expressed in the current formalization. A few examples include *Mutation XSS* (mXSS), *Prototype Pollution* and event-driven executions. In this section, a DOM clobbering example will be used to demonstrate the limits of the formalization. *DOM clobbering* is a DOM XSS exploit where HTML code is injected so that global JavaScript variables can be overruled (i.e. clobbered) using the injected HTML attributes, leading to JavaScript execution [KP23]. DOM clobbering is chosen because it highlights the structural shortcoming of the formalization model.

The Vulnerable Script

Consider the following script:

```

1 // Developer sets a default configuration object
2 var config = { debugMode: false };
3
4 // Attacker injects into the HTML beforehand:
5 // <a id="config" href="javascript:alert(1)"></a>
6
7 // Developer reads from config and writes to the DOM
8 document.getElementById("output").innerHTML = config.href;

```

As shown above, the developer has created a global variable `config` and reads the property `config.href` to write into `innerHTML`. However, if the attacker can insert HTML code into the webpage prior with an `id="config"` attribute (e.g. via stored XSS), the `config` global variable the developer intends to use can be overwritten. Declaring an `href` attribute in the HTML code causes the webpage to execute our JavaScript code: it will resolve to the injected DOM element `<a id="config" href="javascript:alert(1)"></a>`. Hence, reading `config.href` then returns `"javascript:alert(1)"` which is subsequently written to the `innerHTML` sink.

Where the Formalization Breaks Down

Just like in the previous examples, we start with a combined state $\sigma = (g, \rho)$, where $g \in G$ and $\rho : \chi \rightarrow Val$. The formalization implicitly assumes that the DOM grove and the variable environment communicate exclusively through source and sink programs: source programs $\alpha_{src}^{\lambda_{src}}$ read from g to ρ while sink programs $\alpha_{sink}^{\lambda_{sink}}$ write from ρ into g . Φ_g and Φ_ρ are defined as disjoint for this reason, and the transition constraints enforce it: source and flow transitions cannot invalidate any Φ_g proposition, while sink transitions cannot invalidate any Φ_ρ proposition. DOM clobbering violates this assumption. The attacker does not introduce a tainted value into ρ via any source transition $\alpha_{src}^{\lambda_{src}}$. Instead, the attacker mutates g in such a way that a later read operation into ρ resolves into a node-insertion in grove g . The introduction of a tainted value into ρ is not caused by any program – it is caused by the browser binding operation, which cannot be expressed in the formalization.

Failure of the source transition rules In the formalization, a variable $x \in \chi$ acquires a value exclusively through a source transition S-READ or S-DEFAULT, or through a flow transition. Rule S-READ requires an explicit label $\lambda_{src} \in \Lambda_{src}$ identifying the location from which the value originates:

$$\Lambda_{src} \stackrel{\text{def}}{=} \{\text{url}, \text{cookie}, \text{localStorage}, \text{sessionStorage}, \text{dom}, \text{referrer}\}$$

DOM clobbering cannot be expressed with any of these labels. The value of `config.href` does not come from the URL, a cookie, or any storage API. It comes from the grove g via the browser's

implicit binding mechanism, which is not a member of Λ_{src} and is not represented by any atomic program.

Failure of the taint-propagation constraints Because no source transition is used, the taint-propagation constraints never mark `config.href` as tainted. In the formalization, a state in which `config.href` holds the attacker-controlled string `"javascript:alert(1)"` would satisfy:

$$\sigma \in \mathbf{m}_{\mathfrak{R}}(\mathbf{val}_{\text{config.href}}, \text{"javascript:alert(1)"})$$

but would simultaneously satisfy:

$$\sigma \notin \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{\text{config.href}, \lambda}) \quad \forall \lambda \in \Lambda_{src}$$

The variable contains an attacker-controlled value, yet no taint proposition holds. Even when `config.href` is passed to $\alpha_{sink}^{\text{innerHTMLHTML}}$, the sink transition produces a state σ' in which proposition `sinknodeid,innerHTMLHTML` holds but no `tainted` proposition holds simultaneously. The exploit state is defined as:

$$\mathbf{Exploit} \stackrel{\text{def}}{=} \bigcup_{\substack{x \in \chi, \lambda_{src} \in \Lambda_{src} \\ id \in \mathcal{ID}, \lambda_{sink} \in \Lambda_{sink}}} \mathbf{m}_{\mathfrak{R}}(\mathbf{tainted}_{x, \lambda_{src}}) \cap \mathbf{m}_{\mathfrak{R}}(\mathbf{sinknode}_{id, \lambda_{sink}, x})$$

Since the taint intersection is empty, $\sigma' \notin \mathbf{Exploit}$. The formalization thus does not detect the exploit.

Failure of the proposition classes More fundamentally, the separation $\Phi_0 = \Phi_g \cup \Phi_\rho$ has the assumption that the DOM state and the variable environment are independently readable. DOM clobbering introduces a dependency that violates this separation: the value of a variable in ρ is determined by the structure of g , not by any program transition. To express this, you would need propositions of the form:

$$\mathbf{clobbered}_{x, id} \stackrel{\text{def}}{=} \{(g, \rho) \in K \mid \rho(x) = \mathbf{node_ref}(g, id)\}$$

capturing the fact that variable x silently resolves to the node at id in g . Such a proposition crosses the boundary between Φ_g and Φ_ρ and cannot be expressed in either class as currently defined. Introducing it would require modifying both the proposition classes and the transition constraints, since any sink transition that modifies g could now affect the values read through ρ .

7 Discussion

This study aimed to investigate how to formally define Document Object Model cross-site scripting exploits using Propositional Dynamic Logic. To conceptually define this, two state spaces were constructed first: one state space for encapsulating the DOM environment and one state space for encapsulating the JavaScript environment. The DOM environment contained four data types as formal data structures, namely *Trees*, *Forests*, *Groves*, and *Strings*. These data structures can represent the Document Object Model and track its changes. Within JavaScript code, the JavaScript state space is used containing a variable environment. Specifically, this variable environment consists of variable identifiers: standalone variables, list items, field items and storage elements. Program transitions are divided into three semantic roles: source programs $\alpha_{src}^{\lambda_{src}}$, flow programs $\alpha_{flow}^{\lambda_{flow}}$, and sink programs $\alpha_{sink}^{\lambda_{sink}}$, where label λ specifies the type of operation.

To formally define this conceptual framework in Propositional Dynamic Logic, a set of atomic propositions for the DOM grove Φ_g and the variable environment Φ_ρ were defined first. To propagate through states, transitions were constructed. The source transitions specify what type of source operation is performed, and binds a value read from a browser storage location into the variable environment ρ . Then, flow operations (**Declare**, **Lit**, **Assignment**, **Concatenation**, **Trim**, **Sanitization**, **Encoding**, **Decoding**) can be applied. Sink operations are performed at last, writing out the variable from ρ into g . An exploit is reached in case there is a tainted variable with a sinknode present $\mathbf{m}_{\mathcal{R}}(\mathbf{tainted}_{x, \lambda_{src}}) \cap \mathbf{m}_{\mathcal{R}}(\mathbf{sinknode}_{id, \lambda_{sink}, x})$.

The constructed formalization is capable of expressing a wide range of DOM XSS scenarios in PDL. Because of the source operations label λ_{src} , many different sources can be specified, like `url` and `cookie`. Through the flow operations, the intermediate steps between source and sink operations can be expressed, also encompassing temporary modifications to strings like encoding and permanent modifications like sanitization. Using the set of regular expressions $\mathcal{E}$, the sanitization-operations can be specified. Finally, the sink label λ_{sink} allows for different sink transitions, like `innerHTML` and `document.write`, making a wide variety of DOM XSS attacks expressible in the formalization.

Despite this expressivity, the formalization is not all-encompassing. *Prototype Pollution*, for instance, cannot be expressed since the variable environment doesn't allow for JavaScript Prototypes. *Mutation XSS* cannot be expressed either, since the browser's HTML parser may silently rewrite a string into a dangerous tree structure that the sanitizer never saw, whereas the formalization assumes the parse function behaves predictably. *DOM clobbering* is the starkest example highlighting the incomplete formalization, since it exposes a fundamental limitation of the combined state model. The formalization assumes that g and ρ are coupled only by explicit atomic programs in Π_0 , and that taint enters ρ only through labelled source transitions. DOM clobbering violates both assumptions simultaneously: the attacker contaminates ρ by manipulating g through an implicit browser mechanism that is outside Π_0 , and does so without triggering any transition in Λ_{src} . The result is that the formalization assigns no taint to the variable that carries the malicious value, and the exploit state is unreachable in the model even though it is reachable in the browser. Extending the formalization to cover DOM clobbering would require adding a new class of source transitions that model implicit `window[id]`, a corresponding extension of Λ_{src} , and a coupling constraint between Φ_g and Φ_ρ that the current disjoint structure does not admit.

Regarding future work, this formalization could be used for creating an automated analysis tool. Because exploits are expressed as PDL reachability formulas $\sigma \models \langle \alpha_{src}; (\alpha_{flow})^*; \alpha_{sink} \rangle \mathbf{Exploit}$, the question of whether a given script is vulnerable can be assessed in the combined state space. A tool could translate client-side JavaScript into sequences of source, flow, and sink transitions, construct the corresponding Kripke model, and decide reachability of an exploit state automatically. This connects to the original motivation of this thesis, namely the formulation of more precise sanitization rules. By making the conditions for exploitation explicit, the formalization offers a foundation on which the soundness of a proposed sanitizer could be checked against the evasion patterns the model already captures, such as the double-decode bypass of Example 2.

8 Conclusion

This study set out to answer the following research question:

"How can we formally define Document Object Model Cross-site Scripting Exploits using Propositional Dynamic Logic?"

To answer this question, a framework was constructed in which DOM XSS exploits can be expressed as state transitions in Propositional Dynamic Logic. The foundation of this framework is a combined state model, one state space for the Document Object Model, and one for the JavaScript variable environment. The DOM state space is built from four recursive data types. Together, they allow individual nodes to be uniquely identified and tracked as a payload propagates through the document structure. The JavaScript state space represents variables and their values, providing a way to reason about how attacker-controlled data moves through script execution.

From this combined state, two disjoint classes of atomic propositions were derived: one class for the DOM grove and the other class for the variable environment. These proposition classes encode the security-relevant properties of each component and form the basis for expressing when an exploit state is reached. To connect these into a dynamic model, transitions were specified across three semantic classes: (i) source transitions, (ii) flow transitions and (iii) sink transitions. This approach ensures a clean separation between the role of each operation and its effect on the combined state. Taint-propagation constraints were then defined over these transitions. They specify under which conditions a variable remains attacker-controlled as flow transitions are performed. An exploit state is reached when a tainted variable and a sink node are simultaneously present in the same state.

The resulting formalization provides a basis for reasoning about a wide range of DOM XSS scenarios, covering multiple sources, intermediate flow transformations, and multiple sink types within a single unified model.

References

- [Acc21] Accountix. *Acunetix web application vulnerability-report, 2021*. Tech. rep. Accountix, 2021. URL: <https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021/>.
- [BRV02] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002. ISBN: 9780521527149. URL: <https://books.google.nl/books?id=gFEidNVDWVoC>.
- [BV07] Patrick Blackburn and Johan Van Benthem. “1 Modal logic: a semantic perspective”. In: *Studies in logic and practical reasoning*. Vol. 3. Elsevier, 2007, pp. 1–84. DOI: 10.1016/S1570-2464(07)80004-8.
- [FL79] Michael J Fischer and Richard E Ladner. “Propositional dynamic logic of regular programs”. In: *Journal of computer and system sciences* 18.2 (1979), pp. 194–211. DOI: 10.1016/0022-0000(79)90046-1.
- [Gar+08] Philippa A Gardner et al. “Local Hoare reasoning about DOM”. In: *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2008, pp. 261–270. DOI: 10.1145/1376916.1376953.
- [HC96] George Edward Hughes and Maxwell John Cresswell. *A New Introduction to Modal Logic*. London and New York: Routledge (Psychology Press), 1996. ISBN: 9780415125994. DOI: 10.4324/9780203028100.
- [HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000. ISBN: 9780262263023. URL: <https://books.google.nl/books?id=ZRXN0zX15d8C>.
- [Hyd+15] Isatou Hydera et al. “Current state of research on cross-site scripting (XSS)—A systematic literature review”. In: *Information and Software Technology* 58 (2015), pp. 170–186. DOI: 10.1016/j.infsof.2014.07.010.
- [KP23] Soheil Khodayari and Giancarlo Pellegrino. “It’s (dom) clobbering time: Attack techniques, prevalence, and defenses”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1041–1058. DOI: 10.1109/SP46215.2023.10179403.
- [LSJ13] Sebastian Lekies, Ben Stock, and Martin Johns. “25 million flows later: large-scale detection of DOM-based XSS”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 1193–1204. DOI: 10.1145/2508859.2516703.
- [Mel+18] William Melicher et al. “Riding out domsday: Towards detecting and preventing dom cross-site scripting”. In: *2018 Network and Distributed System Security Symposium (NDSS)*. 2018. DOI: 10.14722/NDSS.2018.23309.
- [MMT08] Sergio Maffei, John C Mitchell, and Ankur Taly. “An operational semantics for JavaScript”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2008, pp. 307–325. DOI: 10.1007/978-3-540-89330-1_22.
- [Plo04] Gordon D Plotkin. “A structural approach to operational semantics”. In: *The Journal of Logic and Algebraic Programming* (2004). DOI: 10.1016/j.jlap.2004.05.001.

Appendix

This appendix provides a reference overview of the symbols used throughout this thesis, grouped by category.

Logic and Proof Systems

Symbol	Description
φ, ψ	Meta-variables ranging over formulas
p, q, r	Atomic propositions
Φ_0	Set of atomic propositions
$\neg$	Negation
$\wedge$	Conjunction
$\vee$	Disjunction
$\rightarrow$	Implication
$\leftrightarrow$	Equivalence
0	Falsity (propositional constant)
1	Truth (propositional constant)
$\forall$	Universal quantifier
$\exists$	Existential quantifier
$\models$	Satisfaction relation ($M \models \varphi$: formula φ holds in model M)
$\vdash$	Derivability relation ($\Gamma \vdash \varphi$: φ is derivable from Γ)

Modal and Dynamic Logic

Symbol	Description
$\Box \varphi$	Necessity: φ holds in all accessible worlds
$\Diamond \varphi$	Possibility: φ holds in at least one accessible world
$[\alpha] \varphi$	Box modality: φ necessarily holds after executing program α
$\langle \alpha \rangle \varphi$	Diamond modality: φ possibly holds after executing α ; defined as $\neg[\alpha]\neg\varphi$
$\alpha; \beta$	Sequential composition: execute α then β
$\alpha \cup \beta$	Non-deterministic choice: execute α or β
α^*	Iteration: execute α zero or more times
$\varphi?$	Test: proceed if φ holds, else abort
$a, b, \dots$	Atomic programs; the set of all atomic programs is Π_0
α, β, γ	Complex programs; the set of all programs is Π
Φ	Set of all propositions

Kripke Structures

Symbol	Description
$\mathfrak{F}$	Kripke Frame: a pair $(K, R_{\mathfrak{F}})$
$\mathfrak{K}$	Kripke Model: a triple $(K, R_{\mathfrak{K}}, \mathbf{m}_{\mathfrak{K}})$
K	Universe of states (worlds) in a Kripke structure
$R_{\mathfrak{K}}$	Accessibility (transition) relation on K
R_{λ}	Named transition relation indexed by label λ ; equals $\mathbf{m}_{\mathfrak{K}}(\alpha^{\lambda})$

Symbol	Description
$\mathbf{m}_{\mathfrak{K}}$	Valuation function mapping formulas and programs to subsets of K (or relations on K)
w, v	Individual worlds / states in K
$\mathfrak{K}, w \models \varphi$	Formula φ is satisfied at world w in model $\mathfrak{K}$

DOM State Space

Symbol	Description
T	Set of trees (individual DOM nodes)
F	Set of forests (ordered lists of trees; children of an element)
G	Set of groves (the DOM heap)
S	Set of strings
$t \in T$	A tree, either an element node $s_{id}[A, f]_{fid}$ or a text node $\#text_{id}[s]$
$f \in F$	A forest, built from $\emptyset_F$, $\langle t \rangle_F$, and $f \otimes f$
$g \in G$	A grove, built from $\emptyset_G$, $\langle t \rangle_G$, and $g \oplus g$
$s \in S$	A string, built from $\emptyset_S$, characters c , and concatenation $s \cdot s$
$s_{id}[A, f]_{fid}$	Element node with tag s , node identifier id , attribute map A , child forest f , and forest identifier fid
$\#text_{id}[s]$	Text node with identifier id and string content s
$A : \mathcal{A} \rightharpoonup S$	Attribute map: a partial function assigning string values to attribute names
id	Unique node identifier; $id \in \mathcal{ID}$
fid	Unique forest (child-list) identifier
$\mathcal{ID}$	Universe of node identifiers
$\mathcal{A}$	Universe of attribute names: $\{\text{onerror, onload, onclick, onmouseover, href, src, action}\}$
$\otimes$	Forest composition operator
$\oplus$	Grove composition operator
$\cdot$	String concatenation operator
$\text{nodes}(g)$	Set of node identifiers present in grove g
$g[fid \mapsto f]$	Grove g with the forest at identifier fid replaced by f
$\text{lookup}(g, id)$	Value of node id in grove g

JavaScript State Space

Symbol	Description
ρ	Variable environment: a partial function $\chi \rightharpoonup Val$
χ	Set of variable identifiers; $x \mid x[i] \mid x.f \mid \tau.\chi$
x, y	Variable names. Elements of χ
Val	Set of values (primarily strings in the XSS context)
τ	Storage type; $\tau \in \{\text{url, cookie, localStorage, sessionStorage, dom, referrer}\}$
$\text{dom}(\rho)$	Domain of ρ : the set of variables currently bound
$\rho[x \mapsto s]$	Environment ρ updated so that x maps to s
$\perp$	Unbound value (a declared variable with no value assigned)
ε	Empty string
E	Set of all regular expressions over S

Symbol	Description
$e \in E$	An individual regular expression; $e(s)$ holds if e matches string s
San	Set of all sanitizer instances
$\text{san} \in \text{San}$	A sanitizer instance: a finite set of regular expressions $\{e_1, \dots, e_n\} \subset E$ modelling the patterns a developer intends to block

Combined State Model

Symbol	Description
σ	Combined state: a pair (g, ρ)
σ_0	Initial state (no variables bound, no taint, no sink nodes)
$\sigma \xrightarrow{\alpha^\lambda} \sigma'$	Transition from σ to σ' via program α^λ
$\sigma \sim \sigma'$	States σ and σ' are indistinguishable (agree on all atomic propositions)
Exploit	Set of exploit states, states in which a tainted value has reached a sink node

Atomic Programs and Labels

Symbol	Description
λ	Generic label attached to an atomic program
α^λ	Atomic program carrying label λ
$\alpha_{src}^{\lambda_{src}}$	Source program with label $\lambda_{src} \in \Lambda_{src}$, introduces external input into ρ
$\alpha_{flow}^{\lambda_{flow}}$	Flow program with label $\lambda_{flow} \in \Lambda_{flow}$, transforms values within ρ
$\alpha_{sink}^{\lambda_{sink}}$	Sink program with label $\lambda_{sink} \in \Lambda_{sink}$, reads from ρ and mutates g
Λ_{src}	Set of source labels: <code>{url, cookie, localStorage, sessionStorage, dom, referrer}</code>
Λ_{flow}	Set of flow labels: <code>{declare, lit, assign, concat, trim, sanitize, encode, decode}</code>
Λ_{sink}	Set of sink labels: <code>{innerHTML, outerHTML, document.write, eval, setTimeout, setAttribute}</code>

Atomic Propositions

Symbol	Description
Φ_g	Family of DOM grove propositions ($\Phi_g \subset \Phi_0$)
Φ_ρ	Family of variable-environment propositions ($\Phi_\rho \subset \Phi_0$)
node_{id}	Node id exists in the current grove
$\text{child}_{id_1, id_2}$	Node id_2 is a direct child of node id_1
$\text{attr}_{id, a, s}$	Attribute a of node id has value s
$\text{sinknode}_{id, \lambda_{sink}, x}$	Node id was introduced into the grove in λ_{sink} execution context by the sink that processed variable x

Symbol	Description
bound_x	Variable x is in the domain of ρ
$\text{val}_{x,s}$	Variable x currently holds value s
$\text{tainted}_{x,\lambda_{src}}$	Variable x is tainted by source λ_{src} (carries attacker-controlled data)
$\text{match}_{x,san}$	The value of variable x is matched by at least one expression in sanitizer san

Transition Functions

Symbol	Description
$\text{read} : \Lambda_{src} \rightarrow S$	Partial function retrieving the current value from a storage location
$\text{trim}(side, s, s_{fix})$	Removes one prefix-side occurrence of s_{fix} from s if $side = \text{pre}$, one suffix-side occurrence if $side = \text{suf}$, and returns s otherwise. $side \in \{\text{pre}, \text{suf}\}$
$\text{enc} : S \rightarrow S$	Encoding function (e.g. URL-encoding, HTML-entity encoding, Base64)
$\text{dec} : S \rightarrow S$	Decoding function, counterpart to enc . Applying dec after sanitization may restore dangerous characters that the sanitizer's patterns did not match, which is the basis of double-decode evasion
$\text{parse}_{\lambda_{sink}} : S \rightarrow F \cup G \cup (\mathcal{A} \times S)$	Sink-specific parse function converting a string into a forest with attributes suitable for insertion into the grove; behaviour varies by sink label
$\text{js_parse} : S \rightarrow \Pi$	Parses a string as JavaScript source text and returns the corresponding program (used by the <code>eval</code> rule)
$\text{read_source}(x, id)$	DOM-to-JS interface: reads the content of node id from grove g and binds it to variable x in ρ
$\text{write_sink}(id, x)$	JS-to-DOM interface: parses $\rho(x)$ and replaces the subtree at node id in g with the result

PDL Program Abbreviations

Abbreviation	PDL Definition
skip	$\mathbf{1}?$
fail	$\mathbf{0}?$
if φ then α else β	$\varphi?; \alpha \cup \neg\varphi?; \beta$
while φ do α	$(\varphi?; \alpha)^*; \neg\varphi?$
repeat α until φ	$\alpha; (\neg\varphi?; \alpha)^*; \varphi?$