Post-Dominator Analysis for Precisely Handling Implicit Flows

Abhishek Bichhawat
Saarland University, Germany
Email: bichhawat@cs.uni-saarland.de

***Abstract*—Most web applications today use JavaScript for including third-party scripts, advertisements etc., which pose a major security threat in the form of confidentiality and integrity violations. Dynamic information flow control helps address this issue of information stealing. Most of the approaches over-approximate when unstructured control flow comes into picture, thereby raising a lot of false alarms. We utilize the post-dominator analysis technique to determine the context of the program at a given point and prove that this approach is the most precise technique to handle implicit flows.**

I. Motivation and Related Work

Security violations of confidentiality and integrity due to insecure information flow have risen significantly over the years. Many web applications face these threats due to the inclusion of untrusted third-party scripts and advertisements, which might perform some unwanted operation. One of the solutions to overcome these issues is by *information flow control* (IFC). Information flow can be categorized as *explicit* and *implicit* [5]. While explicit flows arise as a result of direct assignments, implicit flows arise from control dependencies. Implicit flows can also arise due to unstructured control flow and exceptions [3]. It is important to handle implicit flows precisely as over-approximation can lead to rejection of various secure programs as insecure.

Austin and Flanagan [1], [2], show that dynamic analysis is the most effective technique to ensure *termination-insensitive non-interference* [12], a variant of *non-inteference* [7] (which is one of the security properties associated with IFC), in dynamic languages like JavaScript. Austin and Flanagan's work considers a variant of λ-calculus and ignores certain complex features like unstructured control flow. Most of the other works in this area either ignore unstructured control flow or require additional annotations [9], [8]. Besides requiring additional annotations, these approaches are less precise and reject perfectly valid programs. For instance, programs like `l = 0; while (h) { break;} return l;` are rejected, given that `h` is a high variable and `l` is a low variable.

Our approach uses post-dominator analysis to handle implicit flows [4]. Every value in the language has a security label associated with it. These labels are a part of a well-defined security lattice. To handle implicit flows in dynamic IFC, a pc label (program-context label) is maintained, which is an upper bound on the labels of the values that have influenced the control flow thus far. Along with the *no-sensitive-upgrade* (NSU) check [13] or the *permissive-upgrade* strategy [2], this technique helps prevent implicit leaks. For example, in the program `l = 0; if (h) {l = 1;}`, there is an implicit flow from `h` to the final value of `l`. In our approach, the pc label at the assignment `l=1` becomes secret (because of the control dependence on the value in `h`, assuming `h` is labeled secret) and based on whether NSU check or permissive-upgrade check is applied, the assignment is prohibited or `l` is labeled partially leaked. Once the control dependence of `h` ends, it is important that we restore the pc label to its initial state. If not, a perfectly valid assignment would be deemed insecure and subject to the NSU or permissive-upgrade check. To this end, we maintain a stack of pc labels to indicate nesting of control flow in a program. Every time a new control structure is encountered, we push an entry on the stack. When the control influence ends, we pop the entry from the stack. We use the immediate post-dominator (IPD) analysis technique to determine the end of a control structure. We show that the post-dominator analysis is the most precise technique for handling implicit flows due to control structures. This approach is not particular to JavaScript and can apply to dynamic IFC analysis in all languages (including at the bytecode level).

II. Post-dominator Analysis

We maintain a control flow graph (Definition 1) for every function with every instruction being represented as a node. For every branch node (Definition 3), we compute its immediate post-dominator (IPD) (Definition 5) [6], [10], [4]. When executing a branch node, we push a new pc label on the stack *along with* the node's IPD. When we actually reach the IPD, we pop the pc label. We ensure that all the CFGs are intra-procedural and the IPDs of different nodes in the CFG lie in the same function. In our design, every function that may throw an unhandled exception has a special, *synthetic exit node* (`SEN`), which is placed after the regular return node(s) of the function [4]. Every exception-throwing node, whose exception will not be caught within the function, has an outgoing edge to the `SEN`, indicating that the handler is in a previous function. We do *not* push a new pc-stack entry if the IPD of the current node is the same as the IPD on the top of the pc-stack or if the IPD of the current node is the `SEN`, as in this case the *real* IPD, which is outside of this method, is already on the pc-stack. We join the label at the current branch-point with the one on the top of the pc-stack. In fact, the actual IPD of a node having `SEN` as its IPD is the node that is currently on the top of the stack. We show this result in Theorem 2.

Definition 1. Control flow graph (CFG)
A Control Flow Graph *is a directed graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, n_s, n_e, \mathcal{L})$. $\mathcal{N}$ *is the set of nodes.* $\mathcal{E}$ *is the set of control flow edges* $(n_1, n_2) \in \mathcal{E}$, *where* $n_i \in \mathcal{N}$. (n_1, n_2) *represents* n_2 *may immediately execute after* n_1. *The nodes* $n_s, n_e \in \mathcal{N}$ *are special nodes representing the start and end point of* $\mathcal{G}$. *The function* $\mathcal{L}$ *maps the edges in* $\mathcal{E}$ *to a label.*

Definition 2. Path
A path *in a CFG* $\mathcal{G}$ *is a sequence of nodes* $(n_1, n_2, ..., n_m)$ *such that* $(n_i, n_{i+1}) \in \mathcal{E}$, *written as* $n_1 \rightarrow_p n_m$. *A node* n *that lies on the path* $n_1 \rightarrow_p n_m$ *is written as* $n \in n_1 \rightarrow_p n_m$. *The notation* $n_1 < n_2$ *with respect to two nodes* n_1 *and* n_2 *in a CFG* $\mathcal{G}$ *indicates that* n_2 *lies on a path* $n_1 \rightarrow_p n_e$.

Definition 3. Branch-point
A branch-point b *is a node in a CFG* $\mathcal{G}$ *having more than one successor, i.e., outdegree*$(b) > 1$.

Definition 4. Post-dominator
In a CFG $\mathcal{G}$, *a node* n_d *is said to be the* post-dominator *of a node* n *if* $\forall p.n \rightarrow_p n_e$ *passes through* n_d. *We use the notation* n_d `pd` n *to indicate that* n_d *is a post-dominator of* n.

Definition 5. Immediate post-dominator (IPD)
i *is the immediate post-dominator of* $n(IPD(n))$ *iff:*

1) i `pd` n *and*
2) $\nexists n_o \in \mathcal{N}.((n_o$ `pd` $n) \wedge (n_o < i))$ *or* $\forall n_o \in \mathcal{N}.((n_o \neq n) \wedge (n_o$ `pd` $n \Rightarrow n_o$ `pd` $i))$.

III. Results

We show in Theorem 1 that the IPD of a node is the most precise node where the context of an operation can be removed.

Theorem 1 (Precision)**.** *Choosing any node other than the IPD to lower the pc-label will either give unsound results or be less precise.*

Proof. Consider a branch-point $b \in \mathcal{N}$ with IPD $i \in \mathcal{N}$.
Assume that $(n \in \mathcal{N}) \neq i$ is the node where we remove the context of the predicate expression in b. Thus, we have different cases:

- $b < n < i$: Then, $\exists p.n \notin b \rightarrow_p n_e$. Thus, if n performs an action that should not have been performed in the context of the predicate expression in b, it might leak information about the predicate expression in b.
- $b < i < n$: Then, for any $n' \in \mathcal{N}$ such that $i < n' < n$ performing an operation that should not be performed in the context of b would be reported illicit as n' would be executed in the context of b.
  - If n' `pd` b, then $\forall p.n' \in b \rightarrow_p n_e$. Hence, the statement n' executes irrespective of whether the branch at b is taken or not and hence, does not depend on the predicate expression in b, i.e., there is no implicit flow from the predicate expression in b to n', but still the program might be rejected.
  - If n' is a statement executing under the context of another branch-point b', such that b' `pd` b, then as b' does not have any implicit flow from the predicate expression in b, any statement executing under the context of the prediate expression in b' should not be influenced by the context of the predicate expression in b. Hence, the program might be rejected even though there is no information leak.
- $i <> n$: $\forall p.n \notin i \rightarrow_p n_e$ or $\forall p.n \notin b \rightarrow_p n_e$, n will never be reached. Thus, the context of b shall not be removed until n_e such that $b < i < n_e$. Similar reasoning as in the second case with $n = n_e$.

Hence, the most precise node where we can safely remove the context of b is $n = IPD(b) = i$. □

Theorem 2. *The actual IPD of a node having* `SEN` *as its IPD is the node on the top of the pc-stack, which lies in a previously called function.*

Proof. Assume two functions F and G given by the CFGs $\mathcal{G} = (\mathcal{N}, \mathcal{E}, n_s, n_e, \mathcal{L})$ and $\mathcal{G}' = (\mathcal{N}', \mathcal{E}', n'_s, n'_e, \mathcal{L}')$, respectively. The program's start and exit node are given by N_s and N_e, respectively. Consider a branch-point $b' \in \mathcal{N}'$ having `SEN` as its IPD. Assume a branch-point $b \in \mathcal{N}$ such that $b < b' < (i = IPD(b))$, $(i \in \mathcal{N}) \neq$ `SEN`, b is the last executed branch-point and top of the pc-stack contains i. We know that $\forall p.i \in b \rightarrow_p n_e$. Thus, i `pd` b' such that $b < b' < i$. We need to show that $\nexists n \in b' \rightarrow_p i \mid (n$ `pd` $b') \wedge (b' < n < i)$.
Proof by contradiction: Assume $\exists n.n$ `pd` $b' \mid b' < n < i$. Then the node n either lies in the function F or G or in another function H given by the CFG $\mathcal{G}'' = (\mathcal{N}'', \mathcal{E}'', n''_s, n''_e, \mathcal{L}'')$, such that F calls H and H calls G.

- $n \in \mathcal{N}'$: As $IPD(b') =$ `SEN` and `SEN` is the last node in a function($\mathcal{G}'$), $\exists p.n \notin b \rightarrow_p n_e$.
- $n \in \mathcal{N}''$: As $\forall p.n \in b' \rightarrow_p N_e$ and $G() < b'$, thus, $\forall p.n \in G() \rightarrow_p N_e$, which means $IPD(G()) \neq$ `SEN`. Hence, the top of the pc-stack then would have $IPD(G()) = (n'' \in \mathcal{N}'') \leq n$ and not i.
- $n \in \mathcal{N}$: When the call to G or any other function H is made, it would push i, IPD of the branch-point on the top of the pc-stack.

Thus, $\nexists n \in b' \rightarrow_p i \mid (n$ `pd` $b') \wedge (b' < n < i)$. Hence, the top of the pc-stack, i is the actual IPD of any node b' having `SEN` as its intra-procedural IPD. □

IV. Contributions

The results above show that immediate post-dominator analysis technique for handling implicit flows is one of the most precise methods to reduce the number of false alarms in dynamic information flow analysis.

Acknowledgment

This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems (RS3)."

REFERENCES

[1] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, 2009.

[2] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, 2010.

[3] A. Bichhawat. Exception handling for dynamic information flow control. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 718–720, New York, NY, USA, 2014. ACM.

[4] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the 3rd Conference on Principles of Security and Trust*, POST '14, LNCS 8414, pages 159–178, 2014.

[5] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[6] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[8] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th ACM Symposium on Applied Computing*, 2014.

[9] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, 2012.

[10] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 9–18, 2011.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:5–19, 2003.

[12] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[13] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.