On using Substructural types for Static Information Flow Analysis

Machiry Aravind Kumar

UCSB

1 Abstract

Information flow analysis is a type of Data-flow analysis[7] used to determine if there exists a data flow from predefined information sources to predefined sinks. As with any Data-flow analysis, it can de done either statically or dynamically. In this paper, we try to explore the possibility of using substructural types to perform static information flow analysis, specifically for security. We provide reasons that encourage using substructural types for information flow analysis, extensively review existing work and finally present our own Substructural type checker for static information flow analysis.

2 Introduction

Information flow analysis is a very useful Data-flow analysis technique having its applications mostly in security. More specifically, for vulnerability detection, private information leakage and Malware analysis. Information flow analysis when used for vulnerability detection is called Taint tracking or Taint checking. It follows from the observation that any data influenced by user (Tainted Data) poses a potential security risk, if used unverified in the program. Taint tracking is modelled by considering all the sources of user input as information sources a.k.a Taint Source and sensitive operations such as setting some security flag as sinks a.k.a Taint Sinks. Consider the example as shown below:

```
char input[1024];
char processed_data[2048]
...
scanf("%1023s",input);
...
//processed_data is copied from input.
strcpy(processed_data,input);
...
//processed_data under goes more changes.
...
//BAD, this could lead to arbitrary code execution.
execute_sql_query(processed_data);
```

Although, its clear that the code shown in example is bad, but when code is sufficiently complex, its easy to miss these by developer or even by an experienced security auditor.
As mentioned before, Taint tracking can be done dynamically or statically. Although, our work is static, we briefly explain dynamic techniques for completeness. There are many techniques to

perform taint tracking dynamically or during program runtime, which are called Dynamic Taint Tracking techniques. These are usually performed by instrumenting the source code or binary with additional instructions to propagate Taintedness of data and explicitly verifying it at concerned sinks. For each unit or byte of data a bit is used to indicate whether its tainted or not. Taintedness of data is propagated along with it as it moves around the program thus tainting all the data derived from tainted data according to a set of rules, these are generally called transfer functions in Data-flow analysis. As with all dynamic analysis techniques, Dynamic Taint Tracking incurs performance overhead thus not desirable on production systems. For more information on Dynamic Taint Tracking refer [17], [8], [16].
Static Taint tracking [4] uses well known Data-flow techniques to perform taint tracking. Main goal of these techniques is to prove absence of information flow from Taint sources to Taint sinks. Static taint tracking methods can be broadly classified into 2 types: Data-flow based and Type system based. Data-flow based methods piggy back on standard flow-sensitive techniques with custom transfer functions, called Taint propagation rules. Type system based methods model Taint tracking as Type checking problem where type violations indicate possibility of flow from Taint source to sink. Data-flow methods require more work, slower and more precise then type checking based methods. However, its easy to prove correctness of type system based methods and hard to get corresponding implementation wrong.
Most of the type-system based methods use type inference [10] [13], where type informations for the underlying program are determined by inference rules based on user provided types. In this paper, we first motivate the reader on using Substructural type system for Taint Tracking, try to understand existing work and finally propose our own implementation of Type checker for Taint Tracking.

3 Motivation

Substructural type systems provide an elegant way to handle state transitions of resources in a program. Consider basic substructural logic i.e intuitionistic logic without following weakening and contraction rules.

$$\frac{\Gamma \vdash \tau}{\Gamma, \sigma \vdash \tau} \text{ IL-Weakening}$$

$$\frac{\Gamma, \sigma, \sigma \vdash \tau}{\Gamma, \sigma \vdash \tau} \text{ IL-Contraction}$$

This enforces restrictions on usage of known facts. You cannot reuse or ignore facts. There are different variations of these restrictions resulting in different Substructural type systems. Substructural type systems of interest are: Linear type systems, which ensure that every variable is used exactly once and Affine type systems, which ensure that every variable is used utmost once. These can be effectively used to model resource state transition.
Consider the example of simple file operations as shown below:

```
//opens a file from given file name
open(filename): string ==> file:open
//reads a file and returns data
read(file): file:open ==> file:open,string
//closes the file
close(file): file:open ==> file:close
```

Say, if we are interested in ensuring statically that file object will be in correct state when used in corresponding functions. These can be encoded in intuitionistic logic rules as shown below:

$$\frac{\Gamma, filename : string \vdash f = open(filename)}{\Gamma, filename : string \vdash f : open} \text{ open}$$

$$\frac{\Gamma, f : open \vdash contents = read(f)}{\Gamma, f : open \vdash contents : string} \text{ read}$$

$$\frac{\Gamma, f : open \vdash close(f)}{\Gamma \vdash f : close} \text{ close}$$

The above rules clearly specify the state transition of the file object. Its important to note that, not having weakening and contraction makes this feasible. If we add weakening and/or contraction its possible to violate the above properties. To demonstrate, how contraction violates this, assume that contraction is allowed, then it allows us to use derive the following rule from close:

$$\frac{\Gamma, f : open \vdash close(f)}{\Gamma f : open \vdash f : close} \text{ close}$$

Thus violating the required state enforcement.
In general, any stateful analysis can be modelled using Substructural types. Taint tracking, as mentioned before tracks state (Tainted/Not Tainted or Untainted) of variables through out the program and check for state violations. In the initial example, having following rules enables us to use Substructural type checker to perform Static Taint Tracking of the program whose violation prove existence of potential security risk.

$$\frac{\Gamma \vdash scanf(*, input)}{\Gamma \vdash input : Tainted} \text{ scanf}$$

$$\frac{\Gamma, src : Type \vdash strcpy(dst, src)}{\Gamma, src : Type \vdash dst : Type} \text{ strcpy}$$

$$\frac{\Gamma, query : Untainted \vdash execute\_sql\_query(query)}{\Gamma \vdash SafeExecution} \text{ execute\_sql\_query}$$

We belive that above example, clearly suggests Substructural types provide an elegant way to perform Static Taint Tracking. With this motivation, lets see existing work in this area.

4 Existing Work

Researchers have tried using different variations of Substructural types to perform information flow tracking. Although most of the works have tried to solve more general problem, we explore on possibilities of using these directly or indirectly for Taint Tracking.

4.1 A Uniform Type Structure For Secure Information Flow

This work[12] focuses on Traditional Secure Information Flow. Which is the problem of ensuring that secure information should not flow to unauthorized entities. Here secrecy of information and authorization of entities is categorized as lattice based model and information flow should be enforced according to this model. For instance: Secret $\sqsubseteq$ Top Secret which means Secret information can flow to a Top Secret entity but not vice versa.

They define linear/affine typed π-calculus and explain its applicability to secure information flow. To explain this work, We start with brief background of traditional π-calculus with branching constructs, continue with linear/affine typing and additions to perform secure information flow. Finally, conclude with discussion on its applicability to Taint Tracking.

π-calculus. This is a process calculus[14] which provides a formal framework to represent concurrent communication. π-calculus is generic in modelling communication, where not only data but channels themselves can be communicated between process. In other words, a functional programming equivalent for concurrent communication.

Basic entities in π-calculus are *Channels* and *Processes*. Channels are the entities that facilitate communication, they are represented usually by lower case letters: x,y,z,u etc. Process are the entities that use channels to communicate, they are defined in BNF grammar[1] as:

$P ::= x(\overrightarrow{y}).P \mid \; !x(\overrightarrow{y}).P \mid \; \overline{x}\langle\overrightarrow{y}\rangle \mid \; x[\&_{i\in I}(\overrightarrow{y_i}).P_i] \mid \; !x[\&_{i\in I}(\overrightarrow{y_i}).P_i] \mid \; \overline{x}\mathtt{in}_i\langle\overrightarrow{y}\rangle \mid \; P|Q \mid (\nu\ x)P \mid \; 0$

One can understand the grammar better by knowing few conventions:

- $\overrightarrow{y}$ indicates vector of names, where each name could be data or channel.
- For a channel x, $\overline{x}$ is used to indicate sending data to it.
- $(\overrightarrow{y})$ is used to represent data receive which is bound to name $\overrightarrow{y}$, for instance: $x(\overrightarrow{y})$ represents data to be received from channel x and bind to $\overrightarrow{y}$.
- $\langle\overrightarrow{y}\rangle$ is used to represent sending data bound to name $\overrightarrow{y}$, for instance: $\overline{x}\langle\overrightarrow{y}\rangle$ represents data bound to $\overrightarrow{y}$ be sent on channel x.
- Prefix ! indicates replication, which allows to replicate corresponding construct unlimited number of times. Replication is useful to model server, where it continues to listen even after a client connects to it

Following is meaning of different component of the above grammar:

- $x(\overrightarrow{y}).P$ represents input, where input is received from channel x and bound to vector $\overrightarrow{y}$, further continue with Process P.
- $\overline{x}\langle\overrightarrow{y}\rangle$ represent output, where a vector $\overrightarrow{y}$ is sent to channel x.
- $x[\&_{i\in I}(\overrightarrow{y_i}).P_i]$ represents branching construct, where channel x contains several branches and each branch is composed of processes P_i to which y_i is bounded to.
- $\overline{x}\mathtt{in}_i\langle\overrightarrow{y}\rangle$ indicates choosing branch i from channel x (has to be branch) and sending $\overrightarrow{y}$ to it.
- $P|Q$ represent parallel composition of P and Q.
- $(\nu\ x)P$ makes x bounded to P i.e all free occurrences of x in P become private to P.
- 0 represents null operation or termination of the process.

Lets see some examples:

- Consider three parallel components: $(\nu\, x)(\quad \overline{x}\langle z\rangle.0 \quad\mid\quad x(\overrightarrow{y}).\overline{y}\langle x\rangle.x(\overrightarrow{y}).0 \quad) \quad\mid\quad z(\overrightarrow{v}).\overline{v}\langle v\rangle.0.$ Although, we receive vector data($\rightarrow$) lets assume for simplicity that all vectors contain one element which can be referred using vector name. Here, all three components start at same time, but component two and three would be waiting to receive data. whereas component

one will start by sending data(z) on channel x and terminates. Channel x is private to first two components thus component two receives z which gets bounded to $\overrightarrow{y}$, further it sends x on $\overrightarrow{y}$ (which is actually z). Now, component three waiting on z gets data (x) which binds to $\overrightarrow{v}$, it continues by sending v on $\overline{v}$ (i.e x on $\overline{x}$) and terminates. At this point, component two waiting on x will receive data i.e x which gets bounded to $\overrightarrow{y}$ and terminates.

- A server and client could be written as: $!x(\overrightarrow{y}).\overline{y}('server\ response') \quad | \quad \overline{x}\langle z\rangle.z(\overrightarrow{resp}).print(\overrightarrow{resp})$ which gets reduced to: $!x(\overrightarrow{y}).\overline{y}('server\ response')$ and client prints $'server\ response'$. Note the use of replication (!) here to model server. This allows server to replicate its process for each client.

4.1.1 Linear/Affine Typing

As mentioned before Channels and Processes are the only entities in π-calculus. Types for Channels are called *Channel Types* and Types for Processes are called *Action Types*.
Channels Types. Channels types represent properties of a channel, these are defied in terms of *action modes*.
Action modes represent different ways to interact with the channel. These are defined as below:

- **Output modes(p_o)**: These are associated with output (↑,!). These indicate the channel is an output and can be written to. Ex: x in $\overline{x}\langle y\rangle.P$. Possible output modes are: Linear output: $\uparrow_L$, Affine output: $\uparrow_A$, Linear server: $!_L$ and Affine server: $!_A$. Linear output and Affine output are obvious by their names. Linear server will always consume the provided input once, ensuring the response is always sent back (as proof of its consumption) to client, where as Affine server may not send response.

- **Input modes(p_i)**: These modes are associated with input (↓,?). These indicate the channel is input and can be read from. Ex: x in $x(y).P$. Possible input modes are: Linear input: $\downarrow_L$, Affine input: $\downarrow_A$, Client request to Linear server ($!_L$): $?_L$ and Client request to Affine server ($!_A$): $?_A$. Client request modes are different from normal input modes as client request should contain a response channel for server to respond thus different types makes type checking easier.

- **Non Composition mode (↕)**: This indicates that channel is not available for further composition. For instance the channel: $x(v).0|x\langle v\rangle.0$ cannot be composed further thus have mode ↕

Channel types are defined in terms of action modes as:
$\tau ::= \tau_i \mid \tau_o \mid \updownarrow \quad \tau_i ::= (\overrightarrow{\tau})^{p_i} \mid [\&_{j\in I}\overrightarrow{\tau_j}]^{p_i} \quad \tau_o ::= (\overrightarrow{\tau})^{p_o} \mid [\oplus_{j\in I}\overrightarrow{\tau_j}]^{p_o}$
As we can see channel types are composed by nesting, outermost mode of a channel type τ is denoted by $\mathrm{md}(\tau)$ where $\mathrm{md}(\updownarrow) = \updownarrow$. Channel type has typing rules which govern the way channels are formed or *composed*. Typing rules also help in verifying a channel to be well typed, these rules are as defined as follows:
We use following notations to define rules:
$\mathcal{M}_\downarrow = \{\downarrow_L, \downarrow_A\} \quad \mathcal{M}_\uparrow = \{\uparrow_L, \uparrow_A\} \quad \mathcal{M}_! = \{!_L, !_A\} \quad \mathcal{M}_? = \{?_L, ?_A\}$

1. **Basic input/output rule:** Linear or affine input should be composed of client requests so that consumption of this could be notified as response. Formally, $(\overrightarrow{\tau})^p$ with $p \in \mathcal{M}_\downarrow$ is well typed if each $\tau_i \in \overrightarrow{\tau}$ is well typed and $\tau_i \in \mathcal{M}_?$. Similarly, Linear or affine output should be composed of servers.

2. **Linear server/request rules:** Linear server channel should include one and only one Linear/affine output (to which client can provide data to server) and could contain multiple client requests to transmit response or query back for additional arguments. Formally, $(\overrightarrow{\tau})^{!_L}$ is well typed if each $\tau_i \in \overrightarrow{\tau}$ is well typed and there exists unique j s.t. $\text{md}(\tau_j) = \mathcal{M}_{\uparrow}$, and for all other i $\text{md}(\tau_i) \in \mathcal{M}_{?}$. Similarly, Linear client requests should contain one and only one Linear/affine input(from which server can get input) and could contain multiple servers, these act as response channels of corresponding client request. These could be used by processing server to respond back.

3. **Affine server/request rules:** Affine server rules are same as Linear server rules with only restriction that all types should be affine.

Examples of well typed channels: $(()^{\uparrow_L})^{!_L}$, $(()^{\uparrow_A})^{!_A}$, and *not* well typed channels: $(()^{\downarrow_L})^{!_L}$,$(()^{\uparrow_L})^{!_A}$ as they violate last two rules.
Action Types. As mentioned before, action types are types for Process, they are represented as a directed graph with nodes as typed channels i.e $x : \tau$, with no duplicate channel names and edges are represented as $x : \tau \rightarrow y : \tau'$ such that either:
1) $\text{md}(\tau) = \downarrow_L$ and $\text{md}(\tau') = \uparrow_L$, this is input to output, which models client sending input to server
2) $\text{md}(\tau) = !_L$ and $\text{md}(\tau') = ?_L$. this is response from server back to client.

Processes are formed based on typing rules similar to channels, which you can refer from [12].
Secure Information flow: They define a partial order of different secrecy levels. Channels and Processes are now annotated with different secrecy levels. Channel typing and Action typing rules are modified according to the partial order.

4.1.2 Applicability to Taint Tracking

This work could be used for Taint Tracking, where we define Tainted and UnTainted types as secrecy levels and define partial order between them. But a major drawback is the requirement of only Linear/Affine types, this limits the practicality of using this for Taint Tracking. It would be really useful if they had defined π-calculus with both Linear and non-Linear/unrestricted types such that we can just use the unrestricted types with secrecy levels.

4.2 Asynchronous Types

Lot of work has been done on using Substructural types for static verification of distributed protocols/systems [11] [15] [18]. one way to use existing work in this area for Taint Tracking is to model all program statements as asynchronous process where each process expects input from a unique channel and outputs to the channel of the statement in program order.

4.3 Practical Substructural Types for Imperative Programs

This work done at Microsoft Research[9] focusses on using Substructural Types for imperative programs. Linear types have strict requirement that all members of linear type should themselves be linear, they provide a motivating case where this is too strict to be useful. They relax this constraint by defining new primitives called *Adoption* and *Focus* which provide an elegant way to relax this constraint. They implemented their system on a language called *Vault*, which is closer to C. Applicability to Taint Checking needs to be explored as one need to work on modelling Tainted

and UnTainted types using Adoption and Focus in an ingenious way. This is good direction to explore for researchers interested in developing pragmatic systems.
Another practical language that has flavour of Substructural types is Rust[5], refer [6] for documentation and [3] for source code. This is claimed to be used by Mozilla to develop a component of Firefox browser.

5 Laint: Static Taint Tracking using Linear Types

This section presents out implementation of Linear/Affine type checker for Taint Tracking. We created a new language called `myown`, which is a simple imperative C like language and has only 2 types: *taint*(affine type) and *untaint*(unrestricted or normal type). Type *unknown* is used to indicate untyped variables. There is a partial order between these types, as shown below:

$$unknown \sqsubseteq untaint \quad unknown \sqsubseteq taint \quad untaint \sqsubseteq taint$$

The grammar for this language is as shown in Section 6. Typing rules are as follows:

5.0.1 Default Types

These are the default types for variables and constants. Note that unknown is not actually a type, this is to indicate that corresponding variable is not declared or doesn't have a type.

$$\frac{}{\Gamma \vdash v : unknown}\ \text{UntypedVariable}$$

$$\frac{}{\Gamma \vdash int : untaint}\ \text{IntegerConstant}$$

$$\frac{}{\Gamma \vdash string : untaint}\ \text{StringConstant}$$

5.0.2 Variable Declaration (Taint Introduction)

These rules specify how variables can be declared. There is a partial order for the available types as shown below:

$$unknown \sqsubseteq untaint \quad unknown \sqsubseteq taint \quad untaint \sqsubseteq taint$$

Variable can be declared (if initial type is unknown) or redeclared to promote to a higher type in the partial order. This specifies one of the important property of affine types, non-affine types can be converted into affine but not vice versa.

$$\frac{\Gamma, v : t \vdash eval(t1\ v; stmts) \quad t \sqsubseteq t1}{\Gamma, v : t1 \vdash eval(stmts)}\ \text{VariableDeclaration}$$

5.0.3 Binary Expression (Taint Propagation)

Binary expression follow below rules to get their types, as you can see an expression is untainted only if all operands are untainted.

$$\frac{\Gamma \vdash x : taint}{\Gamma \vdash x \bigotimes y : taint} \text{ TaintedBinaryExpression}$$

$$\frac{\Gamma \vdash x : untaint, y : untaint}{\Gamma \vdash x \bigotimes y : untaint} \text{ UnTaintedBinaryExpression}$$

5.0.4 Assignment Expression (Taint Propagation)

Assignment can be done according to the aforementioned partial order.Note that this allows assigning tainted values to untainted variable resulting in tainted the variable.

$$\frac{\Gamma, x : t1, y : t2 \vdash eval(y = x; stmts) \quad t1 \neq unknown \quad t2 \neq unknown \quad t2 \sqsubseteq t1}{\Gamma, x : t1, y : t1 \vdash eval(stmts)} \text{ Assignment}$$

5.0.5 Untainted Control (Taint Policy)

This is one of our taint policy i.e not to allow any tainted value to control program flow. This is captured by below rule which doesn't allow tainted conditional expression in `if` clause.

$$\frac{\Gamma \vdash c : untaint \quad \Gamma \vdash eval(if(c)\{condstmts;\}stmts)}{\Gamma \vdash eval(condstmts; stmts)} \text{ UntaintedCondition}$$

5.0.6 Function Definition evaluation (Taint Propagation)

To evaluative function definition, we follow the below rule, as you can see function body is evaluated by adding types of the arguments and function type itself. whereas statements outside function body are evaluated just by adding function type.

$$\frac{\Gamma \vdash eval(tret\ func(t\ arg1, t2\ arg2, ...)\{funcstmts\}\ stmts)}{\Gamma, arg1 : t, arg2 : t2, func : (t, t2, ...) \rightarrow tret \vdash eval(funcstmts) \quad \Gamma, func : (t, t2, ...) \rightarrow tret \vdash eval(stmts)} \text{ FunctionEvaluvation}$$

5.0.7 Function Declaration evaluation (Taint Propagation)

Similar to function definition, declaration is evaluated by just adding function type to the set of assumptions.

$$\frac{\Gamma \vdash eval(rtype\ foo(t1\ v1, t2\ v2, ...); stmts)}{\Gamma, foo : (t1, t2, ...) \rightarrow rtype \vdash eval(stmts)} \text{ FunctionTypeIntroduction}$$

5.0.8 Evaluating Sanitizer (Taint Policy)

This is an important rule and captures the linearity. Consider **Sanitizers** are functions, which have atleast one parameter of type *taint* and has return type *untaint*. The following rule indicates that a taint variable used as an argument to a sanitizer will be consumed by it and is no longer available for evaluation.

$$\frac{\Gamma, v : taint, sanitizer : (..., taint, ...) \rightarrow untaint \vdash eval(sanitizer(..., v, ...); stmts)}{\Gamma, sanitizer : (..., taint, ...) \rightarrow untaint \vdash eval(stmts)} \text{ TaintConsumption}$$

5.0.9 Function call evaluation with untainted types (Taint Propagation)

As untainted type is non linear type it can be used unlimitedly. The below rules captures this:

$$\frac{\Gamma, v : untaint, foo : (..., t, ...) \to * \vdash eval(foo(..., v, ...); stmts)}{\Gamma, v : untaint, foo : (..., t, ...) \to * \vdash eval(stmts)} \text{ UnTaintPropogation}$$

5.0.10 Non Sanitizer evaluation with tainted types (Taint Propagation)

A non-sanitizer function cannot consume a taint variable as shown in below rule:

$$\frac{\Gamma, v : taint, foo : (..., t, ...) \to taint \vdash eval(foo(..., v, ...); stmts)}{\Gamma, v : taint, foo : (..., t, ...) \to taint \vdash eval(stmts)} \text{ NonSanitizerFunction}$$

Note that there are syntactic rules of the language that are omitted i.e a function cannot be defined multiple times, function signature has to match declaration and definition, variable need to be declared before used. Source code and sample code snippets for this system can be found at [2]

References

[1] BackusNaur Form. `http://en.wikipedia.org/wiki/Backus-Naur_Form`.

[2] Laint Git. `https://github.com/Machiry/myown_typechecker`.

[3] Rust Git. `https://github.com/rust-lang/rust`.

[4] Static Taint Analysis for C. `https://code.google.com/p/tanalysis/`.

[5] The Rust Programming Language. `http://www.rust-lang.org/`.

[6] The Rust Programming Language Doc. `http://doc.rust-lang.org/1.0.0-alpha.2/book/`.

[7] Wikipedia. `http://en.wikipedia.org/wiki/Data-flow_analysis`.

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[9] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Notices*, volume 37, pages 13–24. ACM, 2002.

[10] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. *Flow-sensitive type qualifiers*, volume 37. ACM, 2002.

[11] Simon J Gay and Vasco T Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[12] Kohei Honda and Nobuko Yoshida. *A uniform type structure for secure information flow*, volume 37. ACM, 2002.

[13] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004.

[14] Naoki Kobayashi, Benjamin C Pierce, and David N Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):914–947, 1999.

[15] Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via curry-howard and control. In *ACM Sigplan Notices*, volume 45, pages 39–50. ACM, 2010.

[16] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[17] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[18] Jesse A Tov. *Practical programming with substructural types*. PhD thesis, Northeastern University Boston, 2012.

6 Appendix

myown grammar:

```
translation_unit_or_empty    : translation_unit
                             | empty


translation_unit : external_declaration


translation_unit : translation_unit external_declaration

external_declaration : function_declaration

function_declaration : func_decl SEMI

func_decl : TYPEID ID LPAREN param_list RPAREN

param_list : empty

param_list : declarator
           | declarator COMMA param_list

declarator : TYPEID ID

external_declaration : function_definition

function_definition : func_decl LBRACE statement RBRACE

statement : empty
          | expression_statement statement
          | if_statement statement
```

```
expression_statement : assignment_statement SEMI
                     | func_call SEMI
                     | declarator SEMI
                     | ret SEMI

assignment_statement : id EQUALS expression

id : ID

expression : func_call
           | unary_expression
           | binary_expression
           | cond_expression

binary_expression : unary_expression bin_op unary_expression
                  | unary_expression bin_op binary_expression
                  | binary_expression bin_op binary_expression
                  | LPAREN unary_expression bin_op unary_expression RPAREN
                  | LPAREN unary_expression bin_op binary_expression RPAREN
                  | LPAREN binary_expression bin_op binary_expression RPAREN

unary_expression : id
                 | int_const
                 | func_call

cond_expression : unary_expression GT unary_expression
                | unary_expression LT unary_expression
                | unary_expression EQ unary_expression
                | LPAREN unary_expression GT unary_expression RPAREN
                | LPAREN unary_expression LT unary_expression RPAREN
                | LPAREN unary_expression EQ unary_expression RPAREN
                | cond_expression OR cond_expression
                | NOT cond_expression
                | LPAREN cond_expression OR cond_expression RPAREN
                | NOT LPAREN cond_expression RPAREN

if_statement : IF LPAREN cond_expression RPAREN LBRACE statement RBRACE

bin_op : PLUS
       | MINUS
       | MULTI
       | DIV

func_call : ID LPAREN arg_list RPAREN

arg_list : empty
```

```
arg_list : id
         | int_const
         | str_const
         | id COMMA arg_list
         | int_const COMMA arg_list
         | str_const COMMA arg_list
```