Vulnerability Flow Type Systems

Mohsen Lesani
University of California, Santa Cruz

***Abstract*—Pervasive and critical software systems have dormant vulnerabilities that attackers can trigger and cascade to make the program act as a weird machine. In fact, they harbor exploitable programming models that can be used to compose vulnerabilities and mount attacks. This paper presents a type system to derive the abstract weird machines that programs expose. The type system tracks information flow types to detect vulnerabilities, and abstracts the control flow between vulnerabilities to capture weird machines. We formally prove that the inferred weird machine covers the weird runtime behaviors that the program can exhibit. The resulting machine can then be examined to detect patterns of attacks. An important observation is that attacks are often simple and recurring patterns. We model the abstract machines as regular expressions on vulnerability types. This abstract representation is platform-independent and can be used as a uniform description language for attacks. Further, language inclusion and similar decisions about regular expressions are remarkably more efficient than the same decisions for concrete programs or other formal languages.**

***Index Terms*—Vulnerabilities, Weird Machines, Composed Attacks, Type Systems**

I. Introduction

Modern attacks exploit a long chain of *dormant vulnerabilities* inside deployed functional systems. The *composition* of these vulnerabilities can give attackers powerful capabilities. In fact, these systems seem to harbor programming models that let attackers compose vulnerabilities and mount attacks that appear as *weird machines* [10], [21]. For example, a composition of vulnerabilities such as buffer overflow and code injection for just-in-time compilers can emerge as a weird machine that can execute arbitrary code. This paper puts forward a new venue of investigation for a type theory that tracks the composition of *unintended in addition to intended computation*. Detecting the presence or absence of composed attacks can aid many who strive for more secure software.

Information flow type systems have been used to enforce the correct flow of information. For example, they prevent leaking confidential data, or degrading the integrity or availability of data. They track the types of values that the program manipulates, and let the information flow only if the type of the destination is no less restrictive than the source. However, they do not track vulnerabilities and their composition for higher-level malicious behavior.

This paper presents a type system that derives *abstract weird machines of programs as regular expressions over vulnerability types*. We capture vulnerabilities as effect types. An important observation is that attacks are often *simple and recurring patterns* of vulnerabilities. We model attacks as regular expressions on vulnerability types. This abstract representation is platform-independent and can be used as a uniform description language for exploitable weird machines. Further, language inclusion and other decisions about regular expressions are remarkably more efficient than the same decisions for the concrete program or other formal languages.

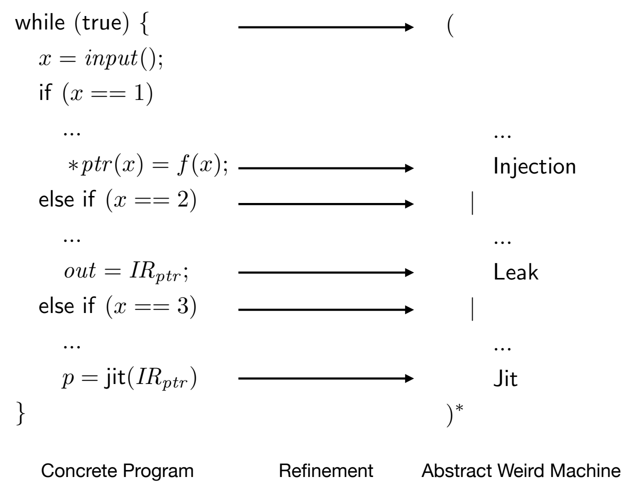

Fig. 1. A simple refinement between the concrete browser program and the DOJITA weird machine.

The type system tracks information flow to capture vulnerabilities present in the program. For example, a buffer overflow is a vulnerability if the user input with low integrity can flow into it. It further tracks the control flow between vulnerabilities. Given a program or a system composed of several components, the type system can infer the present *flow of vulnerabilities* as a regular expression. We prove a refinement relation between the concrete program and the abstract weird machine that the type system derives for it. The weird machine that the type system associates with a program covers the attacks that the executions of that program can exhibit.

The derived weird machine can be used to detect and prevent attacks. Composition is the key to both a successful attack and a successful mitigation: if the abstract program of an attack is exploiting a given sequence of vulnerabilities, sandboxing one vulnerability or reordering their flow can disrupt the attack.

In the following sections, we first consider example attacks that compose multiple vulnerabilities. Then, we present a core language (§ III), its operational semantics (§ IV), and the vulnerability flow type system that can associate abstract weird machines with programs (§ V). We then state and prove the type-safety properties (§ VI). We close the paper with the discussion of related works and conclusion (§ VII and § VIII).

II. Overview

Let us take the DOJITA attack [25] as an example and consider the weird machine that it exploits. In this attack, the adversary injects malicious code into the code section of a browser by leveraging its JIT compilation feature. When a hot function is about to be compiled, the adversary injects its payload into the intermediate representation of the function, and gets it compiled by the JIT compiler. Since the output of the JIT compiler is accepted as safe executable code, the adversary overcomes protections against code injection such as Data Execution Prevention (DEP) or writable xor executable memory (W$\oplus$X).

To perform this attack, the adversary exploits a few vulnerabilities in sequence. First, it exploits a vulnerability to leak the address that the JIT compiler uses to store the IR (intermediate representation) of the function that it compiles. The IR is a syntax tree represented as C++ objects. Second, it exploits a vulnerability to inject a payload containing crafted C++ objects representing the malicious code, and writes it into the IR address. Finally, it uses the JIT compiler to compile the IR of the injected code. The left side of Fig. 1 captures a sketch of a browser as a loop that takes the user input and performs different actions based on that input. In the first branch, the browser contains an injection vulnerability where a value derived from the input is written to an address derived from the input. In the second branch, the browser contains a leak vulnerability where the address of the IR is leaked to a variable that will be visible to the adversary. In the last branch, it compiles the syntax tree stored at the IR pointer.

The concrete browser program provides an abstract weird machine. That machine provides the adversary with a language to compose lurking vulnerabilities, and program attacks such as DOJITA. In Fig. 1, the example browser program on the left provides the abstract weird machine on the right that can be represented as the regular expression $(\mathsf{Injection} \,|\, \mathsf{Leak} \,|\, \mathsf{Jit})^*$. The while statement is abstracted to a Kleene closure, and the if statements are abstracted to alternations. This machine can be used to program many emergent behaviors including the DOJITA attack that is represented as $\mathsf{Leak} \cdot \mathsf{Injection} \cdot \mathsf{Jit}$, i.e., the flow sequence of a leak, an injection, and a JIT compilation. In fact, there is refinement between the concrete program on the left and the abstract program on the right. Any emergent behavior from the captured vulnerabilities of the concrete program is a behavior of the abstract program.

An important observation is that attacks are often compositional, simple and platform independent patterns. In this simple example, we saw that regular expressions can capture vulnerabilities, their composition and patterns of attacks. This abstract representation is platform independent and can be used as a uniform description language for exploitable weird machines.

We will present a vulnerability flow type system that tracks information flow to capture vulnerabilities such as Leak, Jit and $\mathsf{Injection}$ that we saw above, and more importantly associates an abstract weird machine with the concrete program. The abstract weird machine is represented as a regular language that attacks such as DOJITA can be programmed with. The resulting weird machine can be examined to detect the presence and absence of attack patterns. Language inclusion and other decisions about regular expressions are remarkably more efficient than the same decisions for the concrete program or other formal languages such as context-free grammars. For example, given the regular expression $(\mathsf{Injection} \,|\, \mathsf{Leak} \,|\, \mathsf{Jit})^*$ as the weird machine, the complexity of deciding the membership of the DOJITA attack $\mathsf{Leak} \cdot \mathsf{Injection} \cdot \mathsf{Jit}$ is $O(n)$. Further, given the regular expression, the possibility of new classes of unintended behaviors can be examined. Once an attack pattern is found, sandboxing a vulnerability or disrupting an essential control flow can neutralize the the attack pattern in the resulting weird machine. Further, if more expressive languages are necessary to capture particular weird machines, the vulnerability flow type system can be simply adapted to derive abstract programs in those languages.

Next, we first define the syntax of a core language, and then, the operational semantics, and the instrumented operational semantics. Finally, we present the type system and the type-safety theorems.

III. Core Language

Fig. 2 shows the language syntax. An expression e is a value n, a variable x, an operation $e_1 \oplus e_2$, a sequence $e_1; e_2$, a conditional $\mathsf{if}\ e\ e_1\ \mathsf{else}\ e_2$, a loop $\mathsf{while}\ e\ e'$, an assignment $x := e$, or a JIT compilation of an expression jit e. This expression is used to model the just-in-time compilation features of our browser use-case.

The type system associates a weird machine w to a program expression e. We model weird machines as regular expression terms. The alphabet of this language are vulnerability types such as Leak that represents leaking secrets, $\mathsf{Injection}$ that represents injection of payloads into the memory space of the process, and Jit that represents jit compilation (of injected code). A weird machine can be the concatenation $w \cdot w'$ or the alternation $w \,|\, w'$ of two machines w and w', or the Kleene closure w^* of a machine w. These operators can capture the common patterns of vulnerabilities. The void machine is represented as ϵ. A weird machine w is included in another w', written as $w \subseteq w'$, if any instance of the former is an instance of the latter. For example $\mathsf{Leak} \subseteq \mathsf{Leak} \,|\, \mathsf{Injection}$. A weird machine w is a prefix of another w', written as $w \Subset w'$, if any instance of the former is a prefix of an instance of the latter. For example $\mathsf{Leak} \subseteq \mathsf{Leak} \cdot \mathsf{Injection}$.

In order to detect vulnerabilities, the type system associates to each expression e an information flow type f in addition to the weird machine term w. An information flow type f is a tuple $\langle c, i \rangle$ of the confidentiality type c and the integrity type i. The confidentiality and integrity types form lattices $\sqsubseteq$, for example with low L and high H elements. Accordingly, the lattice $\sqsubseteq$ of the flow type f is the product of the two lattices of its elements.

$$
\begin{array}{rcll}
e & := & n \mid x \mid v \mid e_1 \oplus e_2 & \text{Program} \\
 & \mid & e_1; e_2 & \\
 & \mid & \mathsf{if}\ e\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{while}\ e\ e' & \\
 & \mid & x := e \mid \mathsf{jit}\ e & \\
t & := & w, f & \text{Types} \\
w & := & w \cdot w \mid w|w \mid w^* \mid \epsilon & \text{Weird Machine} \\
 & \mid & \mathsf{Leak} \mid \mathsf{Injection} \mid \mathsf{Jit} & \text{Vulnerability Types} \\
f & := & \langle c, i \rangle & \text{Information Flow Type} \\
c & := & \mathsf{L} \mid \mathsf{H} & \text{Confidentiality Type} \\
i & := & \mathsf{L} \mid \mathsf{H} & \text{Integrity Type} \\
v & := & \langle n, f \rangle & \text{Instrumented Value} \\
\mathcal{R} & := & [\,] \mid \mathcal{R} \oplus e \mid v \oplus \mathcal{R} & \text{Reduction Context} \\
 & \mid & \mathcal{R}; e \mid \mathsf{if}\ \mathcal{R}\ e\ \mathsf{else}\ e & \\
 & \mid & x := \mathcal{R} \mid \mathsf{jit}\ \mathcal{R} &
\end{array}
$$

$$
\begin{array}{rcll}
\sigma & := & [\overline{x \mapsto n}] & \text{State} \\
\gamma & := & [\overline{x \mapsto \langle n, f \rangle}] & \text{Instrumented State} \\
\Gamma & := & [\overline{x \mapsto f}] & \text{Type Environment}
\end{array}
$$

Confidentiality Lattice c:
$\bot = \mathsf{L}, \quad \top = \mathsf{H}, \quad \mathsf{L} \sqsubseteq \mathsf{H}$

Integrity Lattice i:
$\bot = \mathsf{H}, \quad \top = \mathsf{L}, \quad \mathsf{H} \sqsubseteq \mathsf{L}$

Product Lattice $\langle c, i \rangle$:
$\bot = \langle \mathsf{L}, \mathsf{H} \rangle, \quad \top = \langle \mathsf{H}, \mathsf{L} \rangle$

Fig. 2. Syntax

IV. Operational Semantics

We first present the operational semantics, and then the instrumented operational semantics that keeps track of the information flow types and further the weird behaviors. We will show the equivalence of the two semantics, and later use the latter to state the type-safety properties.

Operational Semantics. As shown in Fig. 3, the state σ is a mapping from variables to values. The operational semantics defines the relation $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$ that in the pre-state σ, executes the program e for one step, and results in the post-state σ' and the rest of the program e'. The reduction context $\mathcal{R}$ captures the location of the next step in the program.

The rule Var-Sem evaluates a variable x by extracting its value from the store σ. The rule Assn-Sem evaluates an assignment to a variable x by updating the value that it is mapped to in the store σ. The rule Ctx-Sem evaluates the expression e in the reduction context $\mathcal{R}$ without changing the store. The rule Op-Sem evaluates an operation $n_1 \oplus n_2$, and the rule Seq-Sem reduces the sequence v; e, where the first expression is already fully evaluated, to the second expression e. The rules If-Then-Sem and If-Else-Sem reduce the conditional expression to either the then or else expressions depending on whether the condition is non-zero. The rules While-Sem reduces the loop expression $\mathsf{while}\ e\ e'$ by unrolling it once: $\mathsf{if}\ e\ (e'; \mathsf{while}\ e\ e')\ \mathsf{else}\ 0$. The rule Jit-Sem reduces $\mathsf{jit}\ n$ to n, as the result of a JIT-optimized expression stays the same.

Instrumented Operational Semantics. In order to capture the vulnerabilities of the program during execution, we define the instrumented operational semantics. It captures the trace of vulnerabilities that a program execution exhibits. The instrumented state is $\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle$. In order to detect vulnerabilities during assignments to variables, the type environment Γ specifies the expected flow types for each variable. For example, a variable with the flow type $\langle c, i \rangle$ expects at most the confidentiality c, and at least the integrity i. Otherwise, the vulnerabilities Leak and Injection happen respectively. In order to track the information flow types for each variable, the store is instrumented: the instrumented store γ maps each variable x to an instrumented value v that is the pair $\langle n, f \rangle$ of a value n and its flow type f. The instrumented semantics further tracks the implicit (or context) information flow type during the execution, and stores it as f_{x}.

The rule Var-ISem evaluates a variable x by extracting its value from the store γ, and does not incur any vulnerabilities. The rule Assn-ISem evaluates an assignment of a value v to a variable x by updating the value that x is mapped to in the store γ. Further, an assignment can exhibit a Leak vulnerability if the join of confidentiality of the value and the implicit confidentiality cannot flow to the confidentiality of the variable. For example, a value with confidentiality H cannot flow to a variable with confidentiality L. Similarly, an assignment can exhibit an Injection vulnerability if the join of integrity of the value and the implicit integrity cannot flow to the integrity of the variable. For example, a value with integrity L cannot flow to a variable with integrity H. As before, the rule Ctx-ISem evaluates the expression e in the reduction context $\mathcal{R}$ without changing the store. The rule Op-ISem evaluates an operation $n_1 \oplus n_2$, and further calculates the join of the accompanying flow types. As before, the rule Seq-ISem reduces the sequence v; e to e. The rules If-Then-ISem and If-Else-ISem reduce the conditional expression to either the then or else expressions, and further, update the implicit flow type to incorporate the flow type of the condition. The rules While-ISem unrolls the loop as before. The rule Jit-ISem reduces $\mathsf{jit}\ v$ to v, and further, checks that the join of integrity of the value and the implicit integrity is high. Otherwise, JIT is either applied to a low integrity expression, or is called through low integrity control flow that may be controlled by the adversary. In this case, the label captures the Jit vulnerability.

Equivalence. The semantics and the instrumented semantics have tightly related steps: for any step in one, there is a corresponding step in the other one. We formally capture this relation.

$$\textsc{Var-Sem}\quad \langle\sigma, \mathcal{R}[x]\rangle \rightarrow \langle\sigma, \mathcal{R}[\sigma(x)]\rangle$$

$$\textsc{Assn-Sem}\quad \langle\sigma, \mathcal{R}[x := n]\rangle \rightarrow \langle\sigma[x \mapsto n], \mathcal{R}[n]\rangle$$

$$\textsc{Ctx-Sem}\quad \frac{e \rightarrow e'}{\langle\sigma, \mathcal{R}[e]\rangle \rightarrow \langle\sigma, \mathcal{R}[e']\rangle}$$

$$\textsc{Op-Sem}\quad \frac{n_1 \oplus n_2 = n_3}{n_1 \oplus n_2 \rightarrow n_3}$$

$$\textsc{Seq-Sem}\quad v;\ e \rightarrow e$$

$$\textsc{If-Then-Sem}\quad \frac{v \neq 0}{\mathsf{if}\ v\ e_1\ \mathsf{else}\ e_2 \rightarrow e_1}$$

$$\textsc{If-Else-Sem}\quad \mathsf{if}\ 0\ e_1\ \mathsf{else}\ e_2 \rightarrow e_2$$

$$\textsc{While-Sem}\quad \mathsf{while}\ e\ e' \rightarrow \mathsf{if}\ e\ (e';\ \mathsf{while}\ e\ e')\ \mathsf{else}\ 0$$

$$\textsc{Jit-Sem}\quad \mathsf{jit}\ n \rightarrow n$$

Fig. 3. Operational Semantics. $\langle\sigma, e\rangle \rightarrow \langle\sigma, e\rangle$

$$\textsc{Var-ISem}\quad \langle\Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[x]\rangle \rightarrow \langle\Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[\gamma(x)]\rangle$$

$$\textsc{Assn-ISem}\quad \frac{\begin{array}{ccc}\Gamma(x) = \langle c, i\rangle & f_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle & v = \langle n, \langle c', i'\rangle\rangle \\ w_1 = \begin{cases}\epsilon & \text{if } c' \sqcup c_{\mathsf{x}} \sqsubseteq c \\ \mathsf{Leak} & \text{else}\end{cases} & w_2 = \begin{cases}\epsilon & \text{if } i' \sqcup i_{\mathsf{x}} \sqsubseteq i \\ \mathsf{Injection} & \text{else}\end{cases} & w = w_1 \cdot w_2\end{array}}{\langle\Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[x := v]\rangle \xrightarrow{w} \langle\Gamma, \gamma[x \mapsto v], f_{\mathsf{x}}, \mathcal{R}[v]\rangle}$$

$$\textsc{Ctx-ISem}\quad \frac{\langle f_{\mathsf{x}}, e\rangle \xrightarrow{w} \langle f'_{\mathsf{x}}, e'\rangle}{\langle\Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[e]\rangle \xrightarrow{w} \langle\Gamma, \gamma, f'_{\mathsf{x}}, \mathcal{R}[e']\rangle}$$

$$\textsc{Op-ISem}\quad \frac{n_1 \oplus n_2 = n_3 \qquad f_1 \sqcup f_2 = f_3}{\langle f_{\mathsf{x}}, \langle n_1, f_1\rangle \oplus \langle n_2, f_2\rangle\rangle \rightarrow \langle f_{\mathsf{x}}, \langle n_3, f_3\rangle\rangle}$$

$$\textsc{Seq-ISem}\quad \langle f_{\mathsf{x}}, v;\ e\rangle \rightarrow \langle f_{\mathsf{x}}, e\rangle$$

$$\textsc{If-Then-ISem}\quad \frac{n \neq 0}{\langle f_{\mathsf{x}}, \mathsf{if}\ \langle n, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2\rangle \rightarrow \langle f_{\mathsf{x}} \sqcup f'_{\mathsf{x}}, e_1\rangle}$$

$$\textsc{If-Else-ISem}\quad \langle f_{\mathsf{x}}, \mathsf{if}\ \langle 0, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2\rangle \rightarrow \langle f_{\mathsf{x}} \sqcup f'_{\mathsf{x}}, e_2\rangle$$

$$\textsc{While-ISem}\quad \langle f_{\mathsf{x}}, \mathsf{while}\ e\ e'\rangle \rightarrow \langle f_{\mathsf{x}}, \mathsf{if}\ e\ (e';\ \mathsf{while}\ e\ e')\ \mathsf{else}\ 0\rangle$$

$$\textsc{Jit-ISem}\quad \frac{\begin{array}{c} f_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle \qquad v = \langle n, \langle c, i\rangle\rangle \\ w = \begin{cases}\epsilon & \text{if } i \sqcup i_{\mathsf{x}} \sqsubseteq \mathsf{H} \\ \mathsf{Jit} & \text{else}\end{cases}\end{array}}{\langle f_{\mathsf{x}}, \mathsf{jit}\ v\rangle \xrightarrow{w} \langle f_{\mathsf{x}}, v\rangle}$$

Fig. 4. Instrumented Operational Semantics. $\langle\Gamma, \gamma, f_{\mathsf{x}}, e\rangle \xrightarrow{w} \langle\Gamma, \gamma, f_{\mathsf{x}}, e\rangle$

We define the function *pure* that removes the instrumented flow types from an instrumented store.

Definition 1 ($pure(\gamma)$)**.**
$pure([\overline{x \mapsto \langle n, f\rangle}]) \ := \ [\overline{x \mapsto n}]$.

Further, we overload the function *pure* on expressions to remove instrumented flow types from values.

Definition 2 ($pure(e)$)**.**
$pure(\langle n, f\rangle) := n,$
$pure(e_1 \ \oplus \ e_2) := pure(e_1) \ \oplus \ pure(e_2)$
$pure(e_1;\ e_2) := pure(e_1);\ pure(e_2)$
$pure(\mathsf{if}\ e\ e_1\ \mathsf{else}\ e_2) := \mathsf{if}\ pure(e)\ pure(e_1)\ \mathsf{else}\ pure(e_2)$
$pure(\mathsf{while}\ e\ e') := \mathsf{while}\ pure(e)\ pure(e')$
$pure(x := e) := x := pure(e)$
$pure(\mathsf{jit}\ e) := \mathsf{jit}\ pure(e)$

To avoid unnecessary clutter, we leave implicit rewriting of literals n to instrumented literals $\langle n, \bot\rangle$. The instrumented semantics works with instrumented literals; thus, any literal in an expression should be converted to its equivalent instrumented literal before being evaluated by the instrumented semantics.

We can now state the following equivalence theorem. For every execution with the operational semantics, there is a corresponding execution with the instrumented operational semantics, and vice versa.

Theorem 1. *For all* Γ, γ_1, $f_{\mathsf{x}1}$ *and* e_1, *then*
(1) For all γ_2, $f_{\mathsf{x}2}$, e_2 *and* w,
if $\langle\Gamma, \gamma_1, f_{\mathsf{x}1}, e_1\rangle \xrightarrow{w}{}^{*} \langle\Gamma, \gamma_2, f_{\mathsf{x}2}, e_2\rangle$
then $\langle pure(\gamma_1), pure(e_1)\rangle \rightarrow^{*} \langle pure(\gamma_2), pure(e_2)\rangle$.
(2) Further, for all σ_2, e'_2,
if $\langle pure(\gamma_1), pure(e_1)\rangle \rightarrow^{*} \langle\sigma_2, e'_2\rangle$,
then there exists w, γ_2, $f_{\mathsf{x}2}$ *and* e_2 *such that*
$\langle\Gamma, \gamma_1, f_{\mathsf{x}1}, e_1\rangle \xrightarrow{w}{}^{*} \langle\Gamma, \gamma_2, f_{\mathsf{x}2}, e_2\rangle$ *where*
$\sigma_2 = pure(\gamma_2)$ *and* $e'_2 = pure(e_2)$.

Proof. Straightforward by induction on the length of execution, and then case analysis on the step. □

This theorem lets us use the instrumented semantics to state the type-safety theorem in the next sections.

V. Type System

The type system has judgments of the form $\Gamma, f_{\mathsf{x}} \vdash e : w, f$ where Γ is the type environment, f_{x} is the context or implicit information flow type, e is the program expression that is being typed, w is the weird machine of e, and f is the information flow type of e. The judgment is read as follows: under the environment Γ, and the context information flow type f_x, the expression e exposes the abstract weird machine w and has the information flow type f. The type environment Γ maps variables to their flow types f. The context or implicit information flow type f_x represents the flow type of the context under which e is typed, i.e., the type of the enclosing conditions. The type system is presented in Fig. 5.

The rule VAL-TYPE simply type-checks a value n with the void weird machine ϵ and the flow type $\bot$ (that is low confidentiality and high integrity). The rule IVAL-TYPE simply type-checks an instrumented value $\langle n, f\rangle$ with the void weird machine ϵ and the accompanying flow type f. Similarly, the rule VAR-TYPE type-checks a variable x according to the environment Γ.

The rule OP-TYPE type-checks an operation. The resulting weird machine is the concatenation of the weird machines of the operands, and the resulting flow type is join of their flow types. The concatenation operator captures the control flow order of vulnerabilities. Similarly, the rule SEQ-TYPE type-checks a sequence of two expressions. As for operations, the resulting weird machine is the concatenation. However, the resulting flow type is the flow type of the latter operand as the result of a sequence is the result of its second operand.

The rule IF-TYPE type-checks a conditional expression $\mathsf{if}\ e\ e'\ \mathsf{else}\ e''$. The resulting weird machine is $w \cdot (w' \mid w'')$, the concatenation of the weird machine w of the condition e with the alternation of the weird machines w' and w'' of the two branches e' and e''. The alternation captures the fact that the vulnerability of either branch is possible. Each of the two branches e' and e'' are type-checked as f' and f'' under the given context flow type f_x joined with the flow f of the condition e. The resulting flow type is the join of the flow types f' and f'' of the two branches, as the result a conditional can be the result of either of its branches.

The rule WHILE-TYPE type-checks a loop expression $\mathsf{while}\ e\ e'$. It first type-checks the condition e as the weird machine w and flow type f. We note that because the condition e can be recalculated, it is type-checked under the the implicit flow f_x joined with f itself. Then, under the same implicit flow, the rule type-checks the body e' as the weird machine w' and flow type f'. The loop expression is associated with the weird expression $w \cdot (w' \cdot w)^*$ that captures the sequence of w from the condition, and the Kleene closure of the sequence of w' and w from the body and the re-execution of the condition.

The rule ASSN-TYPE type-checks an assignment expression $x \coloneqq e$. Let the context flow type be $\langle c_\mathsf{x}, i_\mathsf{x}\rangle$. The rule first obtains the flow types $\langle c, i\rangle$ and $\langle c', i'\rangle$ for x and e, and the weird machine w for e. It then checks whether the flow is safe. If the join of c' and c_x cannot flow to c, then x may not have enough confidentiality to receive the value of e, and the assignment is associated with a Leak vulnerability w'. Dually, if the join of i' and i_x cannot flow to i, then the value of e may not have enough integrity to be assigned to x, and the assignment is associated with an $\mathsf{Injection}$ vulnerability w''. On the other hand, in both of the above checks, if the flow is legal, the weird machine is void ϵ. The resulting weird machine is the concatenation of the three weird machines w, w' and w''. As the return value of the assignment is the value of x, the resulting flow type is simply the flow type of x.

The rule JIT-TYPE type-checks a JIT expression $\mathsf{jit}\ e$. Let the context flow type be $\langle c_\mathsf{x}, i_\mathsf{x}\rangle$. The rule first type-checks e with the weird machine w and flow type $\langle c, i\rangle$. The rule checks whether the flow to the JIT compiler has high integrity. If the join of i and i_x cannot flow to H, then the passed expression e or the implicit flow leading to the JIT expression may not have enough integrity, and the JIT compilation is associated with a Jit weird machine w'. Otherwise, the weird machine w' is void ϵ. The resulting weird machine is the concatenation of the two weird machines w and w'. As the expression $\mathsf{jit}\ e$ returns the result of compiling e, its flow type is the same as that of e.

The type system tracks information flow to capture vulnerabilities. It can be simple extended to accept vulnerability annotations on the program as well.

VI. TYPE-SAFETY

We now state the type-safety theorem for the type system. We first present a few helper definitions.

We say that an instrumented store γ is consistent with a type environment Γ, written as $\Gamma \vDash \gamma$, if the type of each variable in the store γ flows to its type in the environment Γ.

Definition 3 (Consistency).
$\Gamma \vDash \gamma \ \coloneqq\ \forall (x \mapsto \langle \_, f\rangle) \in \gamma.\ f \sqsubseteq \Gamma(x)$

If a program is typed in an environment Γ, it is executed only with a store γ that is consistent with Γ.

The following preservation lemma states that if an expression e is typed with a weird machine w, then if it steps to an expression e' with a weird behavior w', then w' is included in a prefix w_1 of w. Intuitively, the weird machine w that the type system derives covers any weird step w'. Further, let w_2 be the remainder of w, *i.e.*, $w = w_1 \cdot w_2$. Then, e' is typed with a weird machine that is included in w_2.

Lemma 1 (Preservation). *For all* Γ, f_x, e, w, f, γ, w', γ', f'_x *and* e', *if*
$\Gamma, f_\mathsf{x} \vdash e : w, f$,
$\Gamma \vDash \gamma$, *and*
$\langle \Gamma, \gamma, f_\mathsf{x}, e\rangle \xrightarrow{w'} \langle \Gamma, \gamma', f'_\mathsf{x}, e'\rangle$,
then there exist w'', f', w_1 *and* w_2 *such that*
$\Gamma, f'_\mathsf{x} \vdash e' : w'', f'$,
$w = w_1 \cdot w_2$,
$w' \subseteq w_1$, *and*
$w'' \subseteq w_2$.

The above inclusion property for weird behaviors can be generalized from every step to every execution. If the type system associates a weird machine to a program, that weird machine covers the weird behavior that the executions of the program can exhibit. If the type system type-checks a program

$$\text{VAL-TYPE}\quad \Gamma, f_{\mathsf{x}} \vdash n : \epsilon, \bot$$

$$\text{IVAL-TYPE}\quad \Gamma, f_{\mathsf{x}} \vdash \langle n, f\rangle : \epsilon, f$$

$$\text{VAR-TYPE}\quad \frac{\Gamma(x) = f}{\Gamma, f_{\mathsf{x}} \vdash x : \epsilon, f}$$

$$\text{OP-TYPE}\quad \frac{\Gamma, f_{\mathsf{x}} \vdash e : w, f \qquad \Gamma, f_{\mathsf{x}} \vdash e' : w', f'}{\Gamma, f_{\mathsf{x}} \vdash e \;\oplus\; e' : w \cdot w', f \sqcup f'}$$

$$\text{SEQ-TYPE}\quad \frac{\Gamma, f_{\mathsf{x}} \vdash e : w, f \qquad \Gamma, f_{\mathsf{x}} \vdash e' : w', f'}{\Gamma, f_{\mathsf{x}} \vdash e; e' : w \cdot w', f'}$$

$$\text{IF-TYPE}\quad \frac{\begin{array}{c}\Gamma, f_{\mathsf{x}} \vdash e : w, f \\ \Gamma, f_{\mathsf{x}} \sqcup f \vdash e' : w', f' \qquad \Gamma, f_{\mathsf{x}} \sqcup f \vdash e'' : w'', f''\end{array}}{\Gamma, f_{\mathsf{x}} \vdash \mathsf{if}\ e\ e'\ \mathsf{else}\ e'' : w \cdot (w' \mid w''), f' \sqcup f''}$$

$$\text{WHILE-TYPE}\quad \frac{\begin{array}{c}\Gamma, f_{\mathsf{x}} \sqcup f \vdash e : w, f \\ \Gamma, f_{\mathsf{x}} \sqcup f \vdash e' : w', f'\end{array}}{\Gamma, f_{\mathsf{x}} \vdash \mathsf{while}\ e\ e' : w \cdot (w' \cdot w)^{*}, \bot}$$

$$\text{ASSN-TYPE}\quad \frac{\begin{array}{c}\Gamma(x) = \langle c, i\rangle \qquad \Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle \vdash e : w, \langle c', i'\rangle \\ w' = \begin{cases}\epsilon & \text{if } c' \sqcup c_{\mathsf{x}} \sqsubseteq c \\ \mathsf{Leak} & \text{else}\end{cases} \\ w'' = \begin{cases}\epsilon & \text{if } i' \sqcup i_{\mathsf{x}} \sqsubseteq i \\ \mathsf{Injection} & \text{else}\end{cases}\end{array}}{\Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle \vdash x \coloneqq e : w \cdot w' \cdot w'', \langle c, i\rangle}$$

$$\text{JIT-TYPE}\quad \frac{\begin{array}{c}\Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle \vdash e : w, \langle c, i\rangle \\ w' = \begin{cases}\epsilon & \text{if } i \sqcup i_{\mathsf{x}} \sqsubseteq \mathsf{H} \\ \mathsf{Jit} & \text{else}\end{cases}\end{array}}{\Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}}\rangle \vdash \mathsf{jit}\ e : w \cdot w', \langle c, i\rangle}$$

Fig. 5. Type System. $\Gamma, f \vdash e : w, f$

e as the weird machine w, then any behavior w' that an execution of e exhibits is a prefix of w.

Theorem 2 (Type-safety)**.** *For all* Γ, f_{x}, e w, f, γ, w', γ', f'_{x}, *and* e', *if*

$\Gamma, f_{\mathsf{x}} \vdash e : w, f$,
$\Gamma \vDash \gamma$, *and*
$\langle \Gamma, \gamma, f_{\mathsf{x}}, e\rangle \xrightarrow{w'}{}^{*} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e'\rangle$,

then

$w' \Subset w$.

The above type-safety theorem immediately implies the following corollary. If the type system type-checks a program e as the weird machine w, and w does not intersect with an attack pattern w', then no execution of the program can produce an instance of that attack.

Corollary 2.1. *For all* Γ, f_{x}, e w, f, γ, w', γ', f'_{x}, *and* e', *if*

$\Gamma, f_{\mathsf{x}} \vdash e : w, f$,
$\Gamma \vDash \gamma$,
$w \cap w' = \emptyset$, *and*
$w'' \subseteq w'$,

then

$\langle \Gamma, \gamma, f_{\mathsf{x}}, e\rangle \overset{w''}{\not\to}{}^{*} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e'\rangle$.

In above corollary, the weird behavior w'' is an instance of the attack pattern w'.

VII. Related Works

Discussion. The goal of this paper is to design type systems that derive the abstract weird machines that programs expose. It notes the need for type theories that track unintended in addition to intended behavior of programs, and their composition. Type systems have been applied to check security properties of programs such as non-interference and memory safety. We summarize the relevant works on security type systems, typed assembly language and proof-carrying code below. However, these works do not track vulnerabilities and their composition for higher-level malicious behavior. To the best of our knowledge, this project is the first to present a type system that models vulnerability types and derives the abstract weird machines that the composition of these vulnerabilities can expose. Recently, program logics have been designed to show the incorrectness of programs. These logics can show the presence of bugs; however, they do not consider whether and how these bugs can be exploited, and composed into attacks. Fuzzing is another popular technique that feeds random input to the program to trigger bugs. However, it cannot provide any formal guarantees for the security of the program.

Type Systems. Security type systems [20], [46], [43], [51], [38], [28], [15] have been used to enforce information flow control, and guarantee non-interference. They have been used both to enforce confidentiality and integrity [9], [59] policies. Recently, they have been used to enforce availability and resiliency policies [60], [61], [33] as well. Further, information flow type systems have been used to reason about security properties of composed systems [18], [34] in several domains including concurrent programs [35], app stores [22], and smart contracts [12].

Typed assembly languages [37], [36], [55], [17] model the desired security properties such as control-flow safety as type safety. They design a series of typed intermediate languages and type-preserving transformations between them. Given a well-typed high-level program, they translate the program to a well-typed assembly program. Therefore, the security properties of the high-level program is preserved during the compilation.

Proof-carrying code [40], [8], [23], [52] carries a proof that the code has the desired properties. This principle allows a process to validate the code received from another process efficiently. The sender of the code needs to construct the proof and the receiver can often machine-check the proof with a small trusted computing base in a small amount of time.

The approach has been shown [16] to scale to verifying large programs.

Program Logics. Years after the Hoare-Floyd program logic [27] provided means of proving the correctness of programs, a succession of logics appeared to prove the incorrectness of programs. Reverse [19] and incorrectness [41] logics, its extensions [44], [50], [45], and later its application in practice [32] were based on triples $[p]\, c\, [q]$ that state that for any post-state in q (*i.e.*, incorrect post-states), there is a pre-state in p. Later, reachability logic [39] noted that one incorrect post-state is enough to show a bug, and proposed triples that state the for any pre-state in p, there is a post-state that is in q (*i.e.*, is incorrect). This further provides pre-states that trigger the bug. Later, outcome logic [62] adopted a similar triple, and further presented a unified theory to support both correctness and incorrectness.

Fuzz Testing. Fuzz testing has been a popular testing technique in recent years. It feeds a sequence of random and/or mutated arguments to the program in order to trigger bugs. Many fuzzers have been developed for various software targets, including both user space programs [5], [24], [26], [11], [58], [49], [13], [14], [57], [48] and OS kernels [7], [30], [3], [47], [53], [42], [31], [29], [56]. Fuzzers are recently applied continuously 24/7 [6], [1], [2] and, indeed, have been shown to be effective in finding real-world bugs [4], [54].

VIII. Conclusion

In order to derive the weird machines that programs expose, this paper models vulnerabilities as effect types, and captures the flow between vulnerabilities. It presents a type system that tracks information flow types to detect vulnerabilities such as leak, inject and jit compilation, and abstracts the control flow of vulnerabilities as regular expressions. Both weird machines and composed attacks have simple and recurring patterns, and regular expressions can serve as a uniform platform-independent representation for them. More importantly, language inclusion and intersection that the presence of certain attacks reduce to are efficiently calculated for regular expressions. We formally prove that if the weird machine that the type system infers for a program does not intersect an attack pattern, then the executions of that program are not prone to that attack.

References

[1] ClusterFuzz. https://google.github.io/clusterfuzz/.

[2] OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz/.

[3] Trinity: A Linux System call fuzz tester. https://codemonkey.org.uk/projects/trinity/.

[4] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[5] american fuzzy lop. https://github.com/google/AFL, 2021.

[6] syzbot. https://syzkaller.appspot.com/upstream, 2021.

[7] syzkaller: coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller, 2021.

[8] Andrew W Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256. IEEE, 2001.

[9] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[10] Sergey Bratus, ME Locasto, and ML Patterson. Exploit programming: From buffer overflows to "weird machines" and theory of computation. 2011.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2019.

[12] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. Compositional security for reentrant applications. *arXiv preprint arXiv:2103.08577*, 2021.

[13] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[14] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[15] Stephen Chong and Andrew C Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, 2004.

[16] Christopher Colby, Peter Lee, George C Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[17] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35. Citeseer, 1999.

[18] Anupam Datta, Jason Franklin, Deepak Garg, Limin Jia, and Dilsun Kaynar. On adversary models and compositional security. *IEEE Security & Privacy*, 9(3):26–32, 2010.

[19] Edsko De Vries and Vasileios Koutavas. Reverse hoare logic. In *International Conference on Software Engineering and Formal Methods*, pages 155–171. Springer, 2011.

[20] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20, 1977.

[21] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017.

[22] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104, 2014.

[23] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 67–78, 2007.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[25] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2405–2419, 2017.

[26] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[27] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[28] Sebastian Hunt and David Sands. On flow-sensitive security types. *ACM SIGPLAN Notices*, 41(1):79–90, 2006.

[29] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *40th IEEE Symposium on Security and Privacy*, 2019.

[30] Dave Jones. Triforce linux syscall fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2016.

[31] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: hybrid fuzzing on the linux kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[32] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. Finding real bugs in big programs with incor-

rectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.
[33] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hamraz: Resilient partitioning and replication. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 2267–2284. IEEE, 2022.
[34] Heiko Mantel. On the composition of secure systems. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 88–101. IEEE, 2002.
[35] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 218–232. IEEE, 2011.
[36] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *International Workshop on Types in Compilation*, pages 28–52. Springer, 1998.
[37] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.
[38] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.
[39] Nico Naus, Freek Verbeek, Marc Schoolderman, and Binoy Ravindran. Reachability logic for low-level programs. *arXiv preprint arXiv:2204.00076*, 2022.
[40] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.
[41] Peter W O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.
[42] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor.
[43] Francois Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.
[44] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, pages 225–252. Springer, 2020.
[45] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. A general approach to under-approximate reasoning about concurrent programs. In *34th International Conference on Concurrency Theory (CONCUR 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.
[46] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
[47] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium*, 2017.
[48] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: a neural-network-assisted fuzzer. In *IEEE Security and Privacy*, 2019.
[49] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.
[50] Julien Vanegue. Adversarial logic. In *International Static Analysis Symposium*, pages 422–448. Springer, 2022.
[51] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.
[52] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, 2000.
[53] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *Proc. USENIX Security Symposium*, 2021.
[54] J. Wilk and M. Rash. A collection of vulnerabilities discovered by the AFL fuzzer (afl-fuzz). https://github.com/mrash/afl-cve, 2017.
[55] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 169–180, 2001.
[56] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *41st IEEE Symposium on Security and Privacy*, 2020.
[57] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[58] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
[59] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, 2002.
[60] Lantian Zheng and Andrew C Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 272–286. IEEE, 2005.
[61] Lantian Zheng and Andrew C Myers. A language-based approach to secure quorum replication. PLAS '14, pages 27–39, 2014.
[62] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):522–550, 2023.

IX. Proofs

Theorem 3 (Type Safety). *For all* Γ, f_{x}, e w, f, γ, w', γ', f'_{x}, *and* e', *if*

$\Gamma, f_{\mathsf{x}} \vdash e : w, f$,
$\Gamma \vDash \gamma$, *and*
$\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle \xrightarrow{w'}{}^{*} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e' \rangle$,

then

$w' \Subset w$.

Proof.
We assume
(1) $\Gamma, f_{\mathsf{x}} \vdash e : w, f$
(2) $\Gamma \vDash \gamma$
(3) $\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle \xrightarrow{w'}{}^{*} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e' \rangle$
We show that
$w' \Subset w$.

Let
(4) $w' = w'_1 \cdot ... \cdot w'_n$
(5) $\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle \xrightarrow{w'_1} \langle \Gamma, \gamma_1, f_{\mathsf{x}1}, e_1 \rangle \xrightarrow{w'_2} ... \xrightarrow{w'_n} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e' \rangle$
By induction on the derivation of [3] and Lemma 2, there exists w_1 ... w_n and w_n such that
$w = w_1 \cdot ... \cdot w_n \cdot w_{n+1}$,
$w'_1 \subseteq w_1$, ..., $w'_n \subseteq w_n$
Therefore,
$w' \Subset w$. □

Lemma 2 (Preservation). *For all* Γ, f_{x}, e, w, f, γ, w', γ', f'_{x} *and* e', *if*

$\Gamma, f_{\mathsf{x}} \vdash e : w, f$,
$\Gamma \vDash \gamma$, *and*
$\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle \xrightarrow{w'} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e' \rangle$,

then there exist w'', f', w_1 *and* w_2 *such that*

$\Gamma, f'_{\mathsf{x}} \vdash e' : w'', f'$,
$w = w_1 \cdot w_2$,
$w' \subseteq w_1$, *and*
$w'' \subseteq w_2$.

Proof.
We assume
(1) $\Gamma, f_{\mathsf{x}} \vdash e : w, f$
(2) $\Gamma \vDash \gamma$
(3) $\langle \Gamma, \gamma, f_{\mathsf{x}}, e \rangle \xrightarrow{w'} \langle \Gamma, \gamma', f'_{\mathsf{x}}, e' \rangle$,
We show that there exist f' such that
$\Gamma, f'_{\mathsf{x}} \vdash e' : w'', f'$,
$w = w_1 \cdot w_2$,
$w' \subseteq w_1$, and
$w'' \subseteq w_2$.

The proof is by case analysis on [3]:

Case Var-ISem:
(4) $e = \mathcal{R}[x]$
(5) $\langle \Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[x] \rangle \rightarrow \langle \Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[\gamma(x)] \rangle$
(6) $w' = \epsilon$
By Lemma 3 on [1] and [4],
(7) $\Gamma, f_{\mathsf{x}} \vdash x : w_1, f''$
(8) $w = w_1 \cdot w_2$
By inversion on [7],
(9) $\Gamma(x) = f''$
(10) $w_1 = \epsilon$
By [8] and [10],
(11) $w = w_2$
By [2] on [9], there exists n and f''' such that
(10) $\gamma(x) = \langle n, f''' \rangle$
(11) $f''' \sqsubseteq f''$
By IVal-Type on [10],
(12) $\Gamma, f_{\mathsf{x}} \vdash \gamma(x) : \epsilon, f'''$
By Lemma 4 on [1], [4], [8], [7], [12], and [11], there exists f' such that
(13) $\Gamma, f_{\mathsf{x}} \vdash \mathcal{R}[\gamma(x)] : w, f'$
By [6], [10] and [11], it is straightforward that
(15) $w' \subseteq w_1$
(16) $w \subseteq w_2$
The conclusion is [13], [8], [15] and [16].

Case Assn-ISem:
(4) $e = \mathcal{R}[x \coloneqq v]$
(5) $\Gamma(x) = \langle c, i \rangle$
(6) $f_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}} \rangle$
(7) $v = \langle n, \langle c', i' \rangle \rangle$
(8) $w'_1 = \begin{cases} \epsilon & \text{if } c' \sqcup c_{\mathsf{x}} \sqsubseteq c \\ \mathsf{Leak} & \text{else} \end{cases}$
(9) $w'_2 = \begin{cases} \epsilon & \text{if } i' \sqcup i_{\mathsf{x}} \sqsubseteq i \\ \mathsf{Injection} & \text{else} \end{cases}$
(10) $w' = w'_1 \cdot w'_2$
(11) $\langle \Gamma, \gamma, f_{\mathsf{x}}, \mathcal{R}[x \coloneqq v] \rangle \xrightarrow{w'} \langle \Gamma, \gamma[x \mapsto v], f_{\mathsf{x}}, \mathcal{R}[v] \rangle$
By Lemma 3 on [1] and [4],
(12) $\Gamma, f_{\mathsf{x}} \vdash x \coloneqq v : w_I, f''$
(13) $w = w_I \cdot w_{II}$
By inversion on [12],
(14) $\Gamma(x) = \langle c, i \rangle$
(15) $\Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}} \rangle \vdash v : w_0, \langle c', i' \rangle$
(16) $w_1 = \begin{cases} \epsilon & \text{if } c' \sqcup c_{\mathsf{x}} \sqsubseteq c \\ \mathsf{Leak} & \text{else} \end{cases}$
(17) $w_2 = \begin{cases} \epsilon & \text{if } i' \sqcup i_{\mathsf{x}} \sqsubseteq i \\ \mathsf{Injection} & \text{else} \end{cases}$
(18) $\Gamma, \langle c_{\mathsf{x}}, i_{\mathsf{x}} \rangle \vdash x \coloneqq v : w_0 \cdot w_1 \cdot w_2, \langle c, i \rangle$
(19) $w_I = w_0 \cdot w_1 \cdot w_2$
By inversion on [15],
(20) $w_0 = \epsilon$
By IVal-Type,
(21) $\Gamma, f_{\mathsf{x}} \vdash v : \epsilon, \bot$
Trivially,
(22) $\bot \sqsubseteq f''$
By Lemma 4 on [1], [4], [13], [12], [21], and [22], there exists f' such that
(23) $\Gamma, f_{\mathsf{x}} \vdash \mathcal{R}[v] : w_{II}, f'$
By [8] and [16],

(24) $w'_1 = w_1$
By [9] and [17],
(25) $w'_2 = w_2$
By [13], [19], [20], [24], [25] and [10],
(26) $w = w' \cdot w_{II}$
It is straightforward that
(27) $w' \subseteq w'$, and
(28) $w_{II} \subseteq w_{II}$
The conclusion is [23], [26], [27] and [28].

Case CTX-ISEM:
(4) $e = \mathcal{R}[e_1]$
(5) $\langle f_\times, e_1 \rangle \xrightarrow{w'} \langle f'_\times, e_2 \rangle$
(6) $\langle \Gamma, \gamma, f_\times, \mathcal{R}[e_1] \rangle \xrightarrow{w'} \langle \Gamma, \gamma, f'_\times, \mathcal{R}[e_2] \rangle$
By Lemma 3 on [1] and [4], there exists w_1, w'' and f_1 such that
(7) $\Gamma, f_\times \vdash e_1 : w_1, f_1$
(8) $w = w_1 \cdot w''$
By Lemma 6 on [7] and [5], then there exist w'', f'', w_{11} and w_{12} such that
(9) $\Gamma, f'_\times \vdash e_2 : w_2, f''$
(10) $f'' \sqsubseteq f_1$
(11) $w_1 = w_{11} \cdot w_{12}$
(12) $w' \subseteq w_{11}$
(13) $w_2 \subseteq w_{12}$
By [1], [4], and [8],
(14) $\Gamma, f_\times \vdash \mathcal{R}[e] : w_1 \cdot w'', f$
By Lemma 3 on [14], [7], [9], and [10],
(15) $\Gamma, f'_\times \vdash \mathcal{R}[e_2] : w_2 \cdot w'', f'$
From [8] and [11],
(16) $w = w_{11} \cdot w_{12} \cdot w''$
From [13],
(17) $w_2 \cdot w'' \subseteq w_{12} \cdot w''$
The conclusion is [15], [16], [12] and [17].

□

Lemma 3. *For all* Γ, $f_\times$, $\mathcal{R}$, e, w *and* f, *if*
$\Gamma, f_\times \vdash \mathcal{R}[e] : w, f$,
then there exist w_1, w_2 *and* f_1 *such that*
$\Gamma, f_\times \vdash e : w_1, f_1$, *and*
$w = w_1 \cdot w_2$.

Proof. Straightforward by structural induction on $\mathcal{R}$, and inversion on the typing judgment. □

Lemma 4. *For all* Γ, $f_\times$, $\mathcal{R}$, e_1, w_1, w', f, e_1, f_1, w_2 *and* f_2, *if*
$\Gamma, f_\times \vdash \mathcal{R}[e_1] : w_1 \cdot w', f$,
$\Gamma, f_\times \vdash e_1 : w_1, f_1$,
$\Gamma, f_\times \vdash e_2 : w_2, f_2$, *and*
$f_2 \sqsubseteq f_1$,
then there exist f' *such that*
$\Gamma, f_\times \vdash \mathcal{R}[e_2] : w_2 \cdot w', f'$, *and*
$f' \sqsubseteq f$.

Proof.
We assume
(1) $\Gamma, f_\times \vdash \mathcal{R}[e_1] : w_1 \cdot w', f$
(2) $\Gamma, f_\times \vdash e_1 : w_1, f_1$
(3) $\Gamma, f_\times \vdash e_2 : w_2, f_2$
(4) $f_2 \sqsubseteq f_1$
We show that there exist f' such that
$\Gamma, f_\times \vdash \mathcal{R}[e_2] : w_2 \cdot w', f'$
$f' \sqsubseteq f$

The proof is by structural induction on $\mathcal{R}$:

Case $[\,]$:
From [1],
(5) $\Gamma, f_\times \vdash e_1 : w_1 \cdot w', f$
By Lemma 5 on [5] and [2],
(6) $w' = \epsilon$
(7) $f = f_1$
From [3] and [6],
(8) $\Gamma, f_\times \vdash \mathcal{R}[e_2] : w_2 \cdot w', f_2$
From [4] and [7]
$f_2 \sqsubseteq f$
The conclusion is [8] and [9] with $f' = f_2$.

Case $\mathcal{R} \oplus e$:
From [1],
(5) $\Gamma, f_\times \vdash e_1 + e : w_1 \cdot w', f$
By inversion on [5], there exists w_{11}, f_{11}, w_{12} and f_{12} such that
(6) $\Gamma, f_\times \vdash e_1 : w_{11}, f_{11}$
(7) $\Gamma, f_\times \vdash e : w_{12}, f_{12}$
(8) $\Gamma, f_\times \vdash e_1 \ \oplus \ e : w_{11} \cdot w_{12}, f_{11} \sqcup f_{12}$
(9) $w_1 \cdot w' = w_{11} \cdot w_{12}$
(10) $f = f_{11} \sqcup f_{12}$
By Lemma 5 on [2] and [6],
(11) $w_1 = w_{11}$
(12) $f_1 = f_{11}$
By OP-TYPE on [3] and [7]
(13) $\Gamma, f_\times \vdash e_2 \ \oplus \ e : w_2 \cdot w_{12}, f_2 \sqcup f_{12}$
By [9] and [11],
(14) $w' = w_{12}$
By [13] and [14],
(15) $\Gamma, f_\times \vdash e_2 \ \oplus \ e : w_2 \cdot w', f_2 \sqcup f_{12}$
By [4] and [12],
(16) $f_2 \sqcup f_{12} \sqsubseteq f_{11} \sqcup f_{12}$
By [16] and [10],
(17) $f_2 \sqcup f_{12} \sqsubseteq f$
The conclusion is [15] and [17] with $f' = f_2 \sqcup f_{12}$.

The proof for the other cases $v \ \oplus \ \mathcal{R}$, $\mathcal{R}; e$, if $\mathcal{R}$ e else e, $x := \mathcal{R}$ and jit $\mathcal{R}$ are closely similar to the proof of the case $\mathcal{R} \oplus e$ above. □

Lemma 5. *For all* Γ, f_{x}, $\mathcal{R}$, e, w, f, w' *and* f', *if*

$\Gamma, f_{\mathsf{x}} \vdash \mathcal{R}[e] : w, f$,
$\Gamma, f_{\mathsf{x}} \vdash \mathcal{R}[e] : w', f'$,

then

$w = w'$ *and*
$f = f'$.

Proof. By structural induction on $\mathcal{R}$, and inversion on the two typing judgments. □

Lemma 6. *For all* Γ, f_{x}, e w, f'_{x}, w' *and* e', *if*

$\Gamma, f_{\mathsf{x}} \vdash e : w, f$, *and*
$\langle f_{\mathsf{x}}, e\rangle \xrightarrow{w'} \langle f'_{\mathsf{x}}, e'\rangle$

then there exist w'', f'', w_1 *and* w_2 *such that*

$\Gamma, f'_{\mathsf{x}} \vdash e' : w'', f''$
$f'' \sqsubseteq f$,
$w = w_1 \cdot w_2$,
$w' \subseteq w_1$, *and*
$w'' \subseteq w_2$.

Proof.
We assume
(1) $\Gamma, f_{\mathsf{x}} \vdash e : w, f$
(2) $\langle f_{\mathsf{x}}, e\rangle \xrightarrow{w'} \langle f'_{\mathsf{x}}, e'\rangle$
We show that there exist w'', f'', w_1 and w_2 such that
$\Gamma, f_{\mathsf{x}} \vdash e' : w'', f''$
$f'' \sqsubseteq f$
$w = w_1 \cdot w_2$
$w' \subseteq w_1$
$w'' \subseteq w_2$

Case analysis on [2]:

Case OP-ISEM:
(3) $e = \langle n_1, f_1\rangle \oplus \langle n_2, f_2\rangle$
(4) $w' = \epsilon$
(5) $n_1 \ \oplus \ n_2 \ = \ n_3$
(6) $f_1 \sqcup f_2 = f_3$
(7) $\langle f_{\mathsf{x}}, \langle n_1, f_1\rangle \oplus \langle n_2, f_2\rangle\rangle \rightarrow \langle f_{\mathsf{x}}, \langle n_3, f_3\rangle\rangle$
By [1] and [3],
(8) $\Gamma, f_{\mathsf{x}} \vdash \langle n_1, f_1\rangle \oplus \langle n_2, f_2\rangle : w, f$
By inversion on [8] and its deriving judgments,
(9) $w = \epsilon$
(10) $f = f_1 \sqcup f_2$
By IVAL-TYPE,
(11) $\Gamma, f \vdash \langle n_3, f_3\rangle : \epsilon, f_3$
By [6] and [10],
(12) $f_3 = f$
From [4] and [9], and $w_1 = w_2 = \epsilon$, it is straightforward that
(13) $w = w_1 \cdot w_2$,
(14) $w' \subseteq w_1$
(15) $\epsilon \subseteq w_2$
The conclusion is [11]-[15].

Case IF-THEN-ISEM:
(3) $e = \mathsf{if}\ \langle n, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2$
(4) $w' = \epsilon$
(5) $n \neq 0$
(6) $\langle f_{\mathsf{x}}, \mathsf{if}\ \langle n, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2\rangle \rightarrow \langle f_{\mathsf{x}} \sqcup f'_{\mathsf{x}}, e_1\rangle$
By [1] and [3],
(7) $\Gamma, f_{\mathsf{x}} \vdash \mathsf{if}\ \langle n, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2 : w, f$
By inversion on [7],
(8) $\Gamma, f_{\mathsf{x}} \vdash \langle n, f'_{\mathsf{x}}\rangle : w_0, f_0$
(9) $\Gamma, f_{\mathsf{x}} \sqcup f_0 \vdash e_1 : w_1, f_1$
(10) $\Gamma, f_{\mathsf{x}} \sqcup f_0 \vdash e_2 : w_2, f_2$
(11) $\Gamma, f_{\mathsf{x}} \vdash \mathsf{if}\ \langle n, f'_{\mathsf{x}}\rangle\ e_1\ \mathsf{else}\ e_2 : w_0 \cdot (w_1 \mid w_2), f_1 \sqcup f_2$
(12) $w = w_0 \cdot (w_1 \mid w_2)$
(13) $f = f_1 \sqcup f_2$
By inversion on [8],
(14) $f'_{\mathsf{x}} = f_0$
(15) $w_0 = \epsilon$
From [9] and [14],
(16) $\Gamma, f_{\mathsf{x}} \sqcup f'_{\mathsf{x}} \vdash e_1 : w_1, f_1$
From [13],
(17) $f_1 \sqsubseteq f$
From [12], [15] and [4], and $w_I = \epsilon$, $w_{II} = (w_1 \mid w_2)$, it is straightforward that
(18) $w = w_I \cdot w_{II}$,
(19) $w' \subseteq w_I$
(20) $w_1 \subseteq w_{II}$
The conclusion is [16]-[20],

Case IF-ELSE-ISEM:
Similar to IF-THEN-ISEM.

Case SEQ-ISEM:
Similar to IF-THEN-ISEM.

Case WHILE-ISEM:
(3) $e = \mathsf{while}\ e_1\ e_2$
(4) $w' = \epsilon$
(5) $\langle f_{\mathsf{x}}, \mathsf{while}\ e_1\ e_2\rangle \rightarrow$
$\langle f_{\mathsf{x}}, \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ \mathsf{else}\ 0\rangle$
By [1] and [3],
(6) $\Gamma, f_{\mathsf{x}} \vdash \mathsf{while}\ e_1\ e_2 : w, f$
By inversion on [6],
(7) $\Gamma, f_{\mathsf{x}} \sqcup f \vdash e_1 : w_1, f$
(8) $\Gamma, f_{\mathsf{x}} \sqcup f \vdash e_2 : w_2, f'$
(9) $\Gamma, f_{\mathsf{x}} \vdash \mathsf{while}\ e\ e' : w_1 \cdot (w_2 \cdot w_1)^*, \bot$
(10) $w = w_1 \cdot (w_2 \cdot w_1)^*$
(11) $f = \bot$
By VAL-TYPE,
(12) $\Gamma, f_{\mathsf{x}} \sqcup f \vdash 0 : \epsilon, \bot$
By WHILE-TYPE on [7] and [8],
(13) $\Gamma, f_{\mathsf{x}} \sqcup f \vdash \mathsf{while}\ e_1\ e_2 : w_1 \cdot (w_2 \cdot w_1)^*, \bot$
By SEQ-TYPE on [8] and [13],
(14) $\Gamma, f_{\mathsf{x}} \sqcup f \vdash e_2; \mathsf{while}\ e_1\ e_2 : w_2 \cdot w_1 \cdot (w_2 \cdot w_1)^*, \bot$
that is

(14) $\Gamma, f_\mathsf{x} \sqcup f \vdash e_2; \mathsf{while}\ e_1\ e_2 : (w_2 \cdot w_1)^+, \bot$

By IF-TYPE on [7], [14] and [12],

(15) $\Gamma, f_\mathsf{x} \vdash \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ \mathsf{else}\ 0 :$
$w_1 \cdot ((w_2 \cdot w_1)^+ | \epsilon), \bot$

that is

(15) $\Gamma, f_\mathsf{x} \vdash \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ \mathsf{else}\ 0 :$
$w_1 \cdot (w_2 \cdot w_1)^*, \bot$

Trivially,

(17) $\bot \sqsubseteq f$

From [10], [4], and $w_I = \epsilon$, $w_{II} = w_1 \cdot (w_2 \cdot w_1)^*$, it is straightforward that

(18) $w = w_I \cdot w_{II}$,

(19) $w' \subseteq w_I$

(20) $w_1 \cdot (w_2 \cdot w_1)^* \subseteq w_{II}$

The conclusion is [15]-[20],

Case JIT-ISEM:

(3) $e = \mathsf{jit}\ v$

(4) $f_\mathsf{x} = \langle c_\mathsf{x}, i_\mathsf{x} \rangle$

(5) $v = \langle n, \langle c, i \rangle \rangle$

(6) $w' = \begin{cases} \epsilon & \text{if } i \sqcup i_\mathsf{x} \sqsubseteq \mathsf{H} \\ \mathsf{Jit} & \text{else} \end{cases}$

(7) $\langle f_\mathsf{x}, \mathsf{jit}\ v \rangle \xrightarrow{w'} \langle f_\mathsf{x}, v \rangle$

By [1] and [3],

(8) $\Gamma, f_\mathsf{x} \vdash \mathsf{jit}\ v, f$

By inversion on [8],

(9) $\Gamma, \langle c_\mathsf{x}, i_\mathsf{x} \rangle \vdash v : w_1, \langle c, i \rangle$

(10) $w_2 = \begin{cases} \epsilon & \text{if } i \sqcup i_\mathsf{x} \sqsubseteq \mathsf{H} \\ \mathsf{Jit} & \text{else} \end{cases}$

(11) $\Gamma, \langle c_\mathsf{x}, i_\mathsf{x} \rangle \vdash \mathsf{jit}\ v : w_1 \cdot w_2, \langle c, i \rangle$

(12) $w = w_1 \cdot w_2$

(13) $f = \langle c, i \rangle$

By inversion on [9],

(14) $w_1 = \epsilon$

(15) $\langle c, i \rangle = \bot$

By IVAL-TYPE,

(16) $\Gamma, f_\mathsf{x} \vdash v : \epsilon, \bot$

Trivially,

(17) $\bot \sqsubseteq f$

From [12], [14], [10] and [6],

(18) $w = w'$

From [18], and $w_I = w'$, $w_{II} = \epsilon$, it is straightforward that

(19) $w = w_I \cdot w_{II}$,

(20) $w' \subseteq w_I$

(21) $\epsilon \subseteq w_{II}$

The conclusion is [16], [17], and [19]-[21]. □

Theorem 4. *For all* Γ, γ_1, $f_{\mathsf{x}1}$ *and* e_1, *then*
(1) For all γ_2, $f_{\mathsf{x}2}$, e_2 *and* w,
if $\langle \Gamma, \gamma_1, f_{\mathsf{x}1}, e_1 \rangle \xrightarrow{w}{}^* \langle \Gamma, \gamma_2, f_{\mathsf{x}2}, e_2 \rangle$
then $\langle pure(\gamma_1), pure(e_1) \rangle \to^* \langle pure(\gamma_2), pure(e_2) \rangle$.
(2) Further, for all σ_2, e_2',
if $\langle pure(\gamma_1), pure(e_1) \rangle \to^* \langle \sigma_2, e_2' \rangle$,
then there exists w, γ_2, $f_{\mathsf{x}2}$ *and* e_2 *such that*
$\langle \Gamma, \gamma_1, f_{\mathsf{x}1}, e_1 \rangle \xrightarrow{w}{}^* \langle \Gamma, \gamma_2, f_{\mathsf{x}2}, e_2 \rangle$ *where*
$\sigma_2 = pure(\gamma_2)$ *and* $e_2' = pure(e_2)$.

Proof. Straightforward by induction on the length of execution, and then case analysis on the step. □