Type-Confusion Security

Anonymous authors

***Abstract*—Type systems are a lightweight way to obtain assurance that code enjoys useful properties, including security properties. But in decentralized settings such as smart contracts and federated distributed systems, malicious adversaries may not obey the type system. They may attempt to confuse trusted code by supplying values that do not behave as their advertised type claims. We show that well-known attacks such as Confused Deputy Attacks (CDAs) and reentrancy attacks can be understood as instances of this kind of type confusion. In this paper, we show how to obtain the guarantees of a type system despite the presence of malicious type confusion, using a novel mechanism that combines static and dynamic information flow checking. We give an information-flow-typed core calculus with a run-time enforcement mechanism, and formally prove the security of this mechanism as a hyperproperty-preserving simulation between ill-typed and well-typed programs. With this mechanism, programmers can write code while pretending that adversaries are constrained to obey the type system. The security properties enforced by this ideal system continue to hold in the real system where adversaries lie about types.**

I. Introduction

Many programming languages include mechanisms to enforce isolation and regulate security-critical resources. Some languages (such as JavaScript) rely primarily on run-time mechanisms to enforce security; others (such as Java and Rust) rely heavily on static checking. Even low-level languages such as JVM and WebAssembly rely on static, compile-time checking to enforce security guarantees. Statically checking code for security is an attractive approach to language-based security: of course, it can reduce run-time overhead, but further, a wider range of security properties can be enforced [1]. For example, *noninterference* [2] can be enforced by a static type system.

Unfortunately, purely static enforcement is insufficient in decentralized systems, where untrusted parties need not obey the restrictions imposed by static analysis. Examples of such systems are smart contracts and federated distributed systems with cross-domain information sharing. In such settings, attackers may not respect the typing discipline, and may send values not conforming to types expected by trusted parties, causing *type confusion*.

Moreover, when types themselves are used to express and enforce security policies, type confusion can lead to security vulnerabilities that are particularly challenging to understand and mitigate. For example, we show that *confused deputy attacks (CDA)* [3] can be understood as an instance of type confusion where the type system is used to express access control policies; similarly, *reentrancy attacks* [4] can be understood as a form of type confusion.

A major contribution of this paper is a demonstration of how to enforce type-based security in decentralized systems where adversaries may lie about types and security policies. We give a simple, practical enforcement mechanism that combines static and dynamic checks to provably make type confusion harmless.

To show that our mechanism works, we propose the first formal definition of type-confusion security, inspired by the real–ideal paradigm [5]. In the real world, attackers may lie about types, but in the ideal world, a purely static typing discipline is enforced and attackers respect this discipline. The core result is a proof that with the proposed mechanisms, all real-world executions can be simulated in the ideal world. This result means that programmers can write secure code while assuming a simpler ideal world where the attacker is constrained by types, even though their code really interacts with a stronger adversary who ignores the type system.

The key insight behind our novel enforcement mechanism is to leverage information flow control (IFC). IFC is typically associated with enforcing confidentiality, but it can also be used to enforce integrity, determining which data and control to trust. Some previous languages and systems [6, 7] have used IFC types to enforce security in decentralized environments where adversaries can lie about types, but they have not fully prevented dangerous type confusion. This paper proposes simple mechanisms that fill the gap by going beyond standard IFC.

The rest of the paper is structured as follows.

- §II introduces a motivating example and key intuitions.
- §III provides background for information flow control and gives high-level formal definitions of type confusion.
- §IV defines core calculi λ_{tc}, λ_{tc}^{r} to describe the ideal and real worlds.
- §V formalizes and proves type-confusion security.
- §VI applies our formal results to security proofs.
- §VII discusses related work, and §VIII concludes.

II. Motivation

A. Confused Deputy Attacks

The Confused Deputy Attack (CDA) [3] is a classic—but still not well-understood—security vulnerability in which the attacker tricks a trusted party (called the deputy) into misusing its authority. CDAs have been long described in the literature [3] and mitigation mechanisms exist in systems where a centralized, trusted runtime is available [8], but they continue to cause costly security breaches in modern, decentralized settings such as blockchain smart contracts [9, 10, 11].

The classic example of a CDA [3] is shown in Figure 1. Here, a (paid) compiler service is the confused deputy (a modern equivalent is a cloud-based ML training service). In an honest execution, the user calls the compiler with the source

```
User.main() :=
    Compiler.run{U}(src, outwriter)
```

```
Attacker.main() :=
    Compiler.run{U}(src, billwriter)
```

```
1 Compiler.run{U}(code, outwriter:{U}) :=
2     out, cost := compile{T}(code)
3     outwriter{U}(out)
4     billwriter{T}(cost)
```

Fig. 1: The classic Confused Deputy Attack (CDA) example, annotated with information-flow labels. For simplicity, only function types and function call sites are annotated with pc (control flow) labels.

code and an output file writer; after compiling the source code, the compiler writes compiled code to the output file and records the fee in the billing file. However, an attacker may trick the compiler by passing the billing file itself as the output file, causing the trusted billing file to be overwritten with compiler output.

A traditional defense against CDAs is capability systems [12], in which a trusted entity (such as an operating system) ensures that resources like data and pointers can only be accessed through unforgeable *capability tokens*. A capability system could require that the compiler possess explicit, distinct capabilities for writing to the bill writer and the output file writer, and that the write operation requires sufficiently powerful capabilities to be passed to it. The call to the output file writer should only be passed the capability for the output file—which would not provide enough authority to write to the bill writer. There are two limitations to this approach. First, nothing forces the programmer to select the correct capability for each call, so mistakes are easily still possible. Further, capabilities must be threaded through the computation, so care is required to ensure that they are not accidentally leaked to the attacker; one proposed defense is run-time taint tracking Rajani et al. [8].

To address the limitations of capability systems, we propose a language-based solution in which function types include security annotations. In fig. 1, all functions are annotated with security labels (in braces), where `T` represents "trusted" and `U` represents "untrusted". This annotation, the *external pc (program counter) label*, governs the contexts in which each function may be called. A function labeled `T` may only be called from trusted contexts, but a function labeled `U` may be called from anywhere. We can think of this label as a form of static access control, but unlike in capability systems, information-flow analysis is used to ensure that these security labels are consistent and cannot be chosen arbitrarily.

For example, this consistency implies a subtyping relationship on function types: a `U`-labeled function is a subtype of a `T`-labeled function pointer because it can be used in more contexts. However, the reverse is not true, even though trusted information can be used where untrusted information is

```
User.main() :=
    Compiler.run{U}(src, outwriter)
```

```
1 Attacker.main() :=
2     Compiler.run{U}(src.first(), reenter)
3
4 Attacker.reenter{U◁T}(out) :=
5     outwriter{U}(out)
6     Compiler.run{U}(src.next(), reenter)
```

```
1 Compiler.run{U◁T}(code, outwriter:{U◁U}) :=
2     out, cost := compile{T}(code)
3     outwriter{U}(out)
4     billwriter{T}(cost)
```

Fig. 2: Reentrancy Attacker against the compiler example. In addition to external pc, functions are annotated by a bounding label that bounds the maximum effect during the function execution.

expected. The reason is that function subtyping is contravariant in argument types [13].

This static discipline prevents CDA in an ideal system where all parties respect the typing rules: the attacker is ill-typed because it calls the compiler by passing a `T`-labeled function argument (`billwriter`) to `U`-labeled function parameter (`outwriter`).

However, an attacker who does not respect static typing can mount the original attack; in the real world, run-time checking is needed to prevent CDAs. Perhaps surprisingly, CDAs can be prevented with a simple run-time check based on information flow control. Our new defense is that the caller sends its (statically known) current control-flow label to the callee, and the callee verifies that the caller's control-flow label matches its own label. For example, in fig. 1, the attacker passes `billwriter` as the argument to the compiler. At line 3 of the compiler code, the compiler notifies the callee `billwriter` that it is being called from a `U`-labeled context. As the callee `billwriter` expects calls only from `T`-labeled contexts, it rejects the call at run time.

B. Reentrancy Attack

CDAs are control-flow attacks that exploit unexpected control flow. However, with the typing discipline seen so far, other control-flow attacks remain possible. Figure 2 shows an example of a *reentrancy attack* [14], where the attacker causes the deputy to be called again before it finishes its first call. Consider the attack shown in fig. 2, where the attacker provides a malicious output file writer that calls back to the compiler after writing the output file. This attack turns `Compiler.run` and `Attacker.reenter` into mutually recursive functions that infinitely compile code without billing the user.

As observed by Cecchetti et al. [4], a reentrancy attack happens when functions rely on invariants that do not hold as a result of unexpected control flow. In the compiler example, the output writer is expected to execute at an untrusted level, but its reentrant call to `Compiler.run` happens at a trusted level.

In general, insecure reentrancy happens when the control-flow level moves from a trusted level to an untrusted level and back again to trusted before the first trusted call returns. Such control-flow patterns only happen when calling an *auto-endorsing* function [4]: a function whose external pc label is less trusted than the level at which its code executes. Such functions are essential for expressiveness in decentralized systems. The insight of Cecchetti et al. [4] is that reentrancy attacks, at their core, are caused by unexpected control-flow endorsement via auto-endorsing functions.

We can statically enforce secure reentrancy by annotating functions with a *bounding label* that upper-bounds the level of effects during function execution. A function with an `U` bounding label never endorses its control flow above `U` (subtype); one with a `T` bounding label may call auto-endorsing functions freely (supertype). For example, above, all functions are annotated with a bounding label. The function `Compiler.run` is an auto-endorsing function because it endorses control flow from `U` to `T`, with bounding label equal to `T`. Although `Attacker.reenter` is not auto-endorsing, its bounding label is also `T` because it calls `Compiler.run`; unlike the external pc label, which governs access to the function, the bounding label enforces reentrancy statically by constraining the maximum effect of the function.

Once the bounding label is added to function signatures, the body of the reentrancy attacker becomes ill-typed—it calls the compiler with an ill-typed argument labeled $\{\mathsf{U}\triangleleft\mathsf{T}\}$. Thus, the reentrancy attack is also a form of type confusion.

To prevent reentrancy in the real world with ill-typed attackers, we add a second run-time mechanism, adapting the dynamic lock system of Cecchetti et al. [4]: when a function is called, it locks its own level of control flow until it returns. Any auto-endorsing call that endorses control flow up to or beyond the lock is rejected. In the compiler example, `Compiler.run` locks the `T` level when it is called. `Compiler.run` then calls an untrusted function `Attacker.reenter`, which calls back to `Compiler.run` before the first call to it returns. The dynamic lock prevents this second, reentrant call.

C. Type Confusion and IFC background

In both of the aforementioned attacks, the real world attacker uses type-confused function pointers to cause unintended control flow in trusted code. In the case of CDA, the adversary tricks a trusted party into authorizing an unsafe call using a type-confused function pointer; for reentrancy, the adversary bypasses the invariant on control-flow integrity using a type-confused function pointer. Our key observation is that in security-typed systems, dangerous type confusion arises when trusted code calls type-confused function values. Some classic CDA examples might seem not to fit this description, since they involve writing to references guarded by access control policies. However, a reference can be viewed as a pair of functions: a getter and a setter; with this view, the classic examples can also be understood as involving type-confused functions.

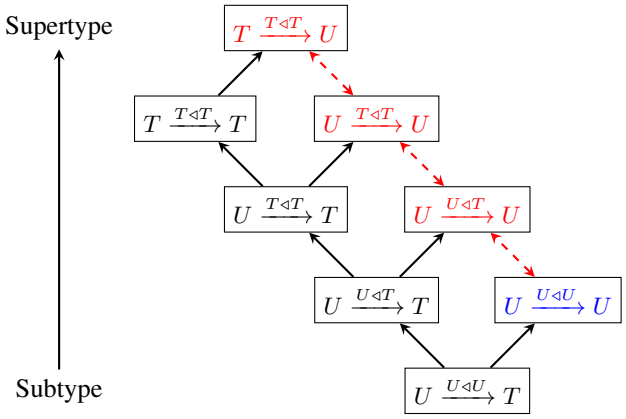

Fig. 3: The subtyping lattice of well-formed function types. The static type of untrusted pointer variables from the attacker have the rightmost type $(U \xrightarrow{U\triangleleft U} U)_U$ (blue). Type confusion happens when the attacker uses pointers with its supertype (red). Type confusing the output type is not harmful because of covariance.

Of course, attackers may also lie about *base types*—the input–output types of functions, and the types of data. However, this base-type confusion is not as challenging as security-type confusion. Base-type confusion can be prevented from compromising security by ensuring that values of the wrong type are treated deterministically as values of the expected type. §V-A discusses this simple defense in more detail.

1) IFC Label Model: We assume a security-typed language in which each type τ includes an information-flow label that governs the use of values of that type. In this paper, we adopt the label model of [15], where the principals $p \in \mathbb{P}$ of the decentralized system generate a lattice of security labels. Given a type τ, we write $\mathsf{lbl}(\tau)$ to denote the label component of the type.

The *flows-to* order $\sqsubseteq$ represents the direction of secure information flow. For simplicity we focus only on enforcing integrity, so $\ell_1 \sqsubseteq \ell_2$ when that ℓ_1 is at least as trusted as ℓ_2. The bottom element $\bot$ represents the most trusted label, and the top element $\top$ represents the most untrusted label. Join $\sqcup$ and meet $\sqcap$ represent the least upper bound and the greatest lower bound of two labels respectively.

2) IFC Function Signatures: Function declarations are annotated with *function types* with the form $\tau_i \xrightarrow{pc_{ex}\triangleleft pc_{top}} \tau_o$. Here, τ_i is the input type and τ_o is the output type. pc_{ex} is the access control label, which we refer to as the *external pc (program counter)*; pc_{top} is the bounding label, which we also refer to as the *top pc*. We believe that declassification of control flow is intrinsically unsafe in decentralized systems, so we only consider the integrity aspect of control flow labels in this work. Like other types, *function pointer types* $\tau = (\tau_i \xrightarrow{pc_{ex}\triangleleft pc_{top}} \tau_o)_\ell$ include an extra label ℓ, which is the label of the function pointer itself.

Auto-endorsing functions, where $pc_{top} \not\sqsubseteq pc_{ex}$, downgrade integrity and therefore explicitly violate noninterference [2] of integrity, which requires that trusted information is not

influenced by untrusted information.

We can understand the possible security-type confusions by exploring the subtyping lattice of function types in an IFC type system. Figure 3 shows the subtyping lattice of function signatures. The output label τ_o is *covariant*—a function that only outputs high-integrity values can be used securely where a function that outputs low-integrity values is expected. The labels τ_i, pc_{ex} and pc_{top} are *contravariant*—a function that can be called in an untrusted context and takes any input may also be called from a trusted context or with a trusted input.

In a decentralized system, an IFC-labeled function type is a security specification. However, certain security specifications are self-contradictory, so certain function types are considered ill-formed. Let p be the function where this function is defined. Then a well-formed function type is one where $pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)$. First, because each function runs at control-flow integrity of p, p is no more trusted than pc_{top}. Second, the external pc should be no more trusted than p so that p may call its own function. Finally, the labels of function inputs should be no more trusted than the external pc pc_{ex}; if not, this function would take trusted input from an untrusted context.

3) Type Confusion of IFC Function Types: With the above background, we are ready to analyze what kind of type confusion may happen to IFC function types. Standard function-call typing rules in prior IFC type systems [16] taint all labels except pc_{top} within the function pointer by the label of the pointer itself.[1] This rule tracks the attacker's influence over the function output through the function pointer. For example, an untrusted pointer to a function with type $(U \xrightarrow{T \lhd T} U)_U$ cannot be called because it requires using an untrusted pointer in a trusted control-flow context.

As reentrancy security forbids control-flow endorsement with the body of trusted functions, calling a function with type $(U \xrightarrow{U \lhd T} U)_U$ from a trusted context is also insecure. As a result, an untrusted function pointer may only be used by a trusted function if all of its type constructors are untrusted.

Visually, in fig. 3, a type confusion is any type cast not permitted by subtyping. As type confusion only happens to function pointers provided by the attacker, such casts only happen to untrusted function pointers. The only type of well-formed untrusted pointer used by trusted principals have the type $(U \xrightarrow{U \lhd U} U)_U$, so there are only three possible down-casts for each contravariant type constructor: the top pc, the external pc and the input type. The run-time checks described previously cover all three cases.

D. The Simple Cases of Type Confusion

1) Nested Type Confusion: A complex case form of type confusion than those in fig. 3 is type confusion of function types whose input and output types are *themselves* function types. This *nested type confusion*, where the attacker lies about the input and output types of function pointers, may cause the deputy to indirectly type-confuse other function pointers. Fortunately, well-formedness of function types ensures that input and output types are untrusted function pointers, whose damage is contained within the untrusted domain.

2) Base Type Confusion: Up to this point, the discussion has focused on confusion of information flow types because such confusion lies at the heart of the most challenging type-confusion vulnerabilities. However, confusion of *base* types can also lead to insecurity in some systems. For example, an adversary might supply an integer where a function pointer is expected, and trusted code might try to call the function, transferring control to an arbitrary memory address. Since such an integer is untrusted, the IFC mechanisms already protect against such confusion, under certain assumptions. Intuitively, when a value v_1 of base type τ_1 is used as if it were a value of base type τ_2, the language run-time must ensure that the behavior resulting from using the value is equivalent to deterministically converting v_2 to *some* value of type τ_2.

III. Formalizing Type Confusion

The examples from §II demonstrate that a dynamic integrity control-flow check and reentrancy locks at remote calls suffices to enforce security against type confusion. In this section, we outline a meta-language for describing decentralized systems and work toward high-level formal definitions of CDA and type confusion inspired by the real–ideal paradigm [5].

A. The Decentralized World Model

In our meta-language, a *program* P is a list of top-level function declarations by different principals p representing different trust domains. A principal may be controlled by an attacker $\mathcal{A}$ that executes arbitrary code. Principals are stateful and each principal has a separate memory heap that maps references to values. Their collective state is a *program context* Σ. We assume (and later define) a deterministic trace semantics for program executions written as $(P, \Sigma, \mathcal{A}) \Downarrow t$. The *behavior* of a program–attacker pair is a function from program context Σ to trace t where $(P, \Sigma, \mathcal{A}) \Downarrow t$.

We conservatively model function pointers as an abstraction of various real-world systems, such as the public methods of EVM smart contracts. First, function pointers are publicly visible and callable. Second, we assume function pointers are *opaque* to the caller: that is, the caller cannot inspect the function body or type without transferring control to the callee. Third, the caller and the callee in remote calls know each other's identities. Fourth, static information flow control is enforced locally within each function. Finally, all functions publish their IFC type signatures, though function types provided by adversarial principals may be lies.

B. Type Confusion Security

Prior static IFC type systems usually either assume well-typed attackers or restrict the interface between trusted code and untrusted code. Type confusion enters the picture when we reason about information flow but consider a more powerful ill-typed attacker. To formalize the idea of *type-confusion*

[1] Most prior IFC systems prove strict noninterference and do not support control flow endorsement. So the external pc is equal to the body ($pc_{ex} = p$) for function types. By the well-formedness condition, the pointer label taints all labels except pc_{top}.

security, we appeal to a simulation argument between well-typed and ill-typed attackers: a system is type-confusion secure when for each ill-typed attacker, there is a well-typed attacker who has the same behavior. This is a kind of real–ideal simulation—the real world has ill-typed attackers but includes run-time checks to ensure that they cannot cause damage; the ideal world has well-typed attackers but does not need run-time checks. Since the real world is just as secure as the ideal world, the programmer can think about security in terms of the ideal world with its well-behaved attackers.

Definition III.1 (Type Confusion Security)**.** A program P is secure against type confusion when for every ill-typed attacker $\mathcal{A}^r$ running in the real world with run-time checks, there exists a well-typed attacker $\vdash \mathcal{A}^i$ that produces the same behavior in the ideal-world dynamic semantics with no run-time checks. With real-world execution and ideal-world execution represented by the notations $\Downarrow_r$ and $\Downarrow_i$, security can be formalized concisely:

$$\forall \mathcal{A}^r \ \exists \vdash \mathcal{A}^i \ \forall \Sigma$$
$$(P, \Sigma, \mathcal{A}^r) \Downarrow^r t \wedge (P, \Sigma, \mathcal{A}^i) \Downarrow^i t' \implies t \approx_{\bar{A}} t'$$

This definition ensures *robust hyperproperty preservation* (RHP) [17], which means that all hyperproperties that hold in the well-typed setting are preserved even in the presence of ill-typed attackers.

Hyperproperties are relational properties over multiple traces of programs that can express complex security and correctness requirements. For example, both noninterference [2] and nonmalleable information flow (NMIF) [18] are hyperproperties that relate 2 and 4 traces respectively. RHP thus enables sound decentralized-world security reasoning assuming a well-typed attacker model.

Type confusion security depends on the semantics of both the well-typed and ill-typed languages. In order for the definition to be useful, the ideal world (well-typed world) should be easier to reason about than the real world (ill-typed world); the ideal world should be expressive enough to build interesting programs, and the real world should be realistic enough to model practical attacks.

In the following sections, we instantiate two IFC-typed core calculi. The ideal world calculus has no run-time checks and only well-typed attackers; the real world calculus has run-time checks that causes well-typed programs to abort when they are type-confused. We then instantiate type confusion security for the calculi, and use it to prove preservation of all security hyperproperties between the ideal and the real world.

IV. A CORE CALCULUS FOR TYPE CONFUSION

In this section, we define λ_{tc}, a formal model of a low-level decentralized language that with remote function pointers. λ_{tc} uses both static and dynamic information flow control to enforce security.

A. Syntax

Figure 4 shows the syntax of λ_{tc}. The language models a decentralized system with multiple principals $p \in \mathbb{P}$, a subset

Principal	$p, q \in \mathbb{P} \subseteq \mathbb{L}$
Label	$\ell, pc \in (\mathbb{L}, \sqsubseteq)$
Bool	$b \in \{\mathsf{true}, \mathsf{false}\}$
Variable	$x \in \mathbb{X}$
Fun. Name	$f \in \mathbb{F}$
Ref Name	$h \in \mathbb{H}$
FnDecl	$F \in \mathbb{P} \times \mathbb{F} \to \mathsf{Signatures} \times \mathsf{Functions}$
HeapType	$R \in \mathbb{P} \times \mathbb{H} \to \mathsf{Types}$
Program	$P \in \mathsf{FnDecl} \times \mathsf{HeapType}$
Adv. Lbl.	$A \subseteq \mathbb{P}$
Adv. Fun.	$M(A) \in A \times \mathbb{F} \to \mathsf{Signatures} \times \mathsf{Functions}$
Attacker	$\mathcal{A} \in A \times M(A)$
Heap	$\Sigma \in \mathbb{P} \times \mathbb{H} \to \mathsf{Values}$

Functions	F	::=	$p.f : \tau \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau := \lambda x.e$
Ref. Decl.	R	::=	$p.h : \tau$
Value	v	::=	$() \mid b_\ell \mid p.f_\ell \mid x$
Expr.	e	::=	$v \mid \mathsf{abort} \mid v \otimes v \mid v\ [pc]v \mid \{e\}_{p.f}$
		$\mid$	$v \downarrow \ell \mid \mathsf{write}_h(v) \mid \mathsf{read}_h$
		$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e$
Signature	η	::=	$\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o$
Type	τ	::=	$1 \mid \mathsf{bool}_\ell \mid (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell \mid 0$
Event	w	::=	$\epsilon \mid (p.h := v) \mid \mathsf{call}^{pc}(p) \mid \mathsf{ret}(p) \mid \bot$
Trace	t	::=	$\epsilon \mid t; w$

Fig. 4: Syntax of λ_{tc}.

of which are controlled by an attacker A. Here, a program $P = (F, R)$ is a pair of a function declaration table F and a heap type declaration table R. The function declaration table F maps each function name $p.f$ to its static type signature and function body. At run time, the attacker function table $M(A)$ replaces the bodies of functions defined on attacker-controlled principals. As the function bodies are statically type-checked against the published function signatures from F, the attacker cannot lie about function signatures. The program also contains a heap declaration table R, which maps heap names $p.h$ to their static type signatures. Σ is the run-time heap that maps heap names to values. Equivalently, the function and heap table can be characterized by lists of function/heap declarations with distinct names.

Function bodies have the form $\lambda x.e$, where x is the input variable and e is the function body expression. An expression may be a value v, which is either a unit, a boolean, a function pointer, or a variable. Expressions in λ_{tc} follow A-Normal Form (ANF) [19]: each intermediate computation is bound to a variable and control flow only happens at if-expressions, let-bindings and remote function applications. Computation includes aborting (abort), binary operations between bools ($v \otimes v$), remote function applications ($v\ [pc]v$), returning ($\{e\}_p$), information downgrading ($v \downarrow \ell$) and reading and writing the heap (read_h,$\mathsf{write}_h(v)$). Each remote function application is tagged with the pc label used for the remote call, which in a real language would be inferred statically. Wrappers $\{e\}_{p.f}$ are not part of the surface syntax and are

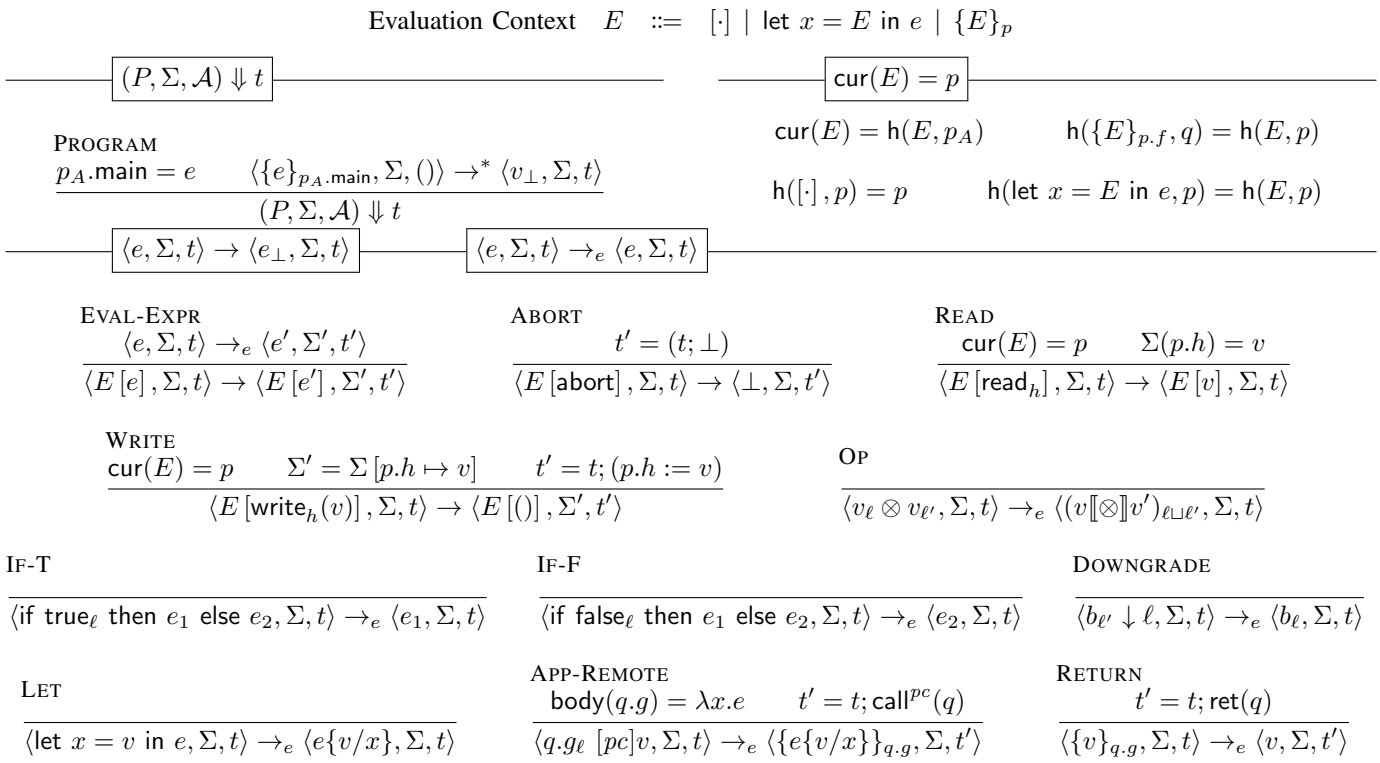

Fig. 5: Dynamic Semantics of λ_{tc}.

used to track the execution context of expressions. Information downgrading expressions are controlled declassification and endorsement operations [20] that explicitly violate information flow policies for necessary expressiveness. Write expressions are the only expressions that cause side effects to the heap.

Reads and writes access the heap locations h at the principal the expression runs on. In other words, λ_{tc} has no expressions for remote read and write; however, remote read and write could be implemented by reader and writer functions. This allows us to treat references and function pointers uniformly, for simplicity.

The type system uses standard information flow control (IFC) types [21] explained in §II-C, where function signatures are annotated with input type, output type, external pc and top pc. Function $\mathsf{lbl}(\tau)$ function that returns the label component of a type τ; unit values are always trusted ($\mathsf{lbl}(1) = \bot$).

The program emits trace events w during execution. An event may be a write event $(p.h := v)$ of a value v, a call event $\mathsf{call}^{pc}(p)$ to principal p, a return event $\mathsf{ret}(p)$ from principal p or an abort event $\bot$.

B. Attackers

Formally, each attacker is a dependent pair $\mathcal{A} = (A, M(A))$, where $A \subseteq \mathbb{P}$ is the set of principals the attacker controls. We assume the existence of an attacker-controlled principal p_A who trusts all other attackers (p_A has integrity level $\top$). The attacker function table contains a special main function $p_A.\mathsf{main}$ that serves as the entry point of a program.

Let the program be $P = (F, R)$, attacker be $\mathcal{A} = (A, M(A))$. We define a helper functions $\mathsf{body}(p.f)$ that dynamically returns the body of the function after attacker replacement.

$$\mathsf{body}(p.f) = \begin{cases} \lambda x.e & p \in A \ \wedge \ M(A)(p.f) = (\eta, \lambda x.e) \\ \lambda x.e & p \notin A \ \wedge \ F(p.f) = (\eta, \lambda x.e) \end{cases}$$

C. Dynamic Semantics

The dynamic semantics of λ_{tc} is shown in fig. 5. The program starts at the attacker's main function and either terminates with a value or aborts. The run-time configuration keeps track of the heap Σ, the current expression e and a trace t of events emitted so far. We make more note of remote application, return, abort, read and write. In APP-REMOTE, the call expression steps as usual by the body of the function with the parameter replaced by the argument. However, to keep track of the execution context of the remote call, the body is wrapped by braces annotated with the function identifier. RETURN operationally removes wrapper braces, effectively going back to the caller context. The helper function $\mathsf{cur}()$ over evaluation context finds the innermost wrapper braces and identifies the principal of the function currently executing. ABORT steps directly to a special value $\bot$, terminating the program immediately. READ reads from the heap Σ and WRITE writes to the heap and emits an event.

$\boxed{\vdash P}$ $\boxed{P \vdash \mathcal{A}}$ $\boxed{P \vdash \Sigma}$

T-SIGNATURE
$$\frac{F(p.f) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e) \qquad x : \tau_i; p; (p, pc_{top}) \vdash e : \tau \qquad pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)}{\vdash (F, R)}$$

T-ADV
$$\frac{\mathcal{A} = (A, M(A)) \qquad \vdash (M(A), R)}{(F, R) \vdash \mathcal{A}}$$

T-STORE
$$\frac{R(p.h) = \tau \qquad \Sigma(p.h) = v \qquad \emptyset; pc; p; pc_{top}^{self} \vdash v : \tau'}{(F, R) \vdash \Sigma}$$

$\boxed{\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau}$

T-UNIT
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash () : 1}$$

T-BOOL
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash b_\ell : \mathsf{bool}_\ell}$$

T-FIELD
$$\frac{F(q.g) = (\eta, \lambda x.e)}{\Gamma; pc; p; pc_{top}^{self} \vdash q.g_\ell : \eta_\ell}$$

T-ABORT
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{abort} : \tau}$$

T-VAR
$$\frac{}{\Gamma, x : \tau; pc; p; pc_{top}^{self} \vdash x : \tau}$$

T-OP
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v_i : \mathsf{bool}_{\ell_i}}{\Gamma; pc; p; pc_{top}^{self} \vdash v_1 \otimes v_2 : \mathsf{bool}_{\ell_1 \sqcup \ell_2}}$$

T-DOWNGRADE
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \mathsf{bool}_{\ell'}}{\Gamma; pc; p; pc_{top}^{self} \vdash v \downarrow \ell : \mathsf{bool}_\ell}$$

T-LET
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau' \qquad \Gamma, x : \tau'; pc; p; pc_{top}^{self} \vdash e : \tau}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}$$

T-IF
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \mathsf{bool}_\ell \qquad \Gamma; pc \sqcup \ell; p; pc_{top}^{self} \vdash e_i : \tau_i}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1 \sqcup \tau_2}$$

T-READ
$$\frac{R(p.h) = \tau \qquad pc \sqsubseteq \mathsf{lbl}(\tau)}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{read}_h : \tau}$$

T-WRITE
$$\frac{R(p.h) = \tau \qquad \Gamma; pc; p; pc_{top}^{self} \vdash v : \tau \qquad pc \sqsubseteq \mathsf{lbl}(\tau)}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{write}_h(v) : 1}$$

T-APP
$$\frac{\begin{array}{c}\Gamma; pc; p; pc_{top}^{self} \vdash v_1 : (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell \qquad \Gamma; pc; p; pc_{top}^{self} \vdash v_2 : \tau \\ \tau \preccurlyeq \tau_i \qquad pc \sqcup \ell \sqsubseteq pc_{ex} \qquad pc_{ex} \sqsubseteq pc_{top} \sqcup p \qquad pc_{top}^{self} \sqsubseteq pc_{top}\end{array}}{\Gamma; pc; p; pc_{top}^{self} \vdash v_1\ [pc_{ex}]v_2 : \tau_o \sqcup \ell}$$

T-WRAPPER
$$\frac{F(q.g) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e) \qquad x : \tau_i; q; q; pc_{top} \vdash e : \tau_o}{\Gamma; pc'; p; pc_{top}^{self} \vdash \{e\}_{q.g} : \tau_o}$$

T-SUB
$$\frac{\tau' \preccurlyeq \tau \qquad \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau'}{\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau}$$

$\boxed{\tau \preccurlyeq \tau}$

S-UNIT
$$\frac{}{\tau \preccurlyeq 1}$$

S-EMPTY
$$\frac{}{0 \preccurlyeq \tau}$$

S-BOOL
$$\frac{\ell \sqsubseteq \ell'}{b_\ell \preccurlyeq b_{\ell'}}$$

S-FUNC
$$\frac{\ell \preccurlyeq \ell' \qquad \tau_o \preccurlyeq \tau_o' \qquad \tau_i' \preccurlyeq \tau_i \qquad pc_{ex}' \preccurlyeq pc_{ex} \qquad pc_{top}' \preccurlyeq pc_{top}}{(\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell \preccurlyeq (\tau_i' \xrightarrow{pc_{ex}' \triangleleft pc_{top}'} \tau_o')_{\ell'}}$$

Fig. 6: Static semantics of λ_{tc}.

D. Static Semantics

Figure 6 shows the static semantics of λ_{tc}. As explained in §II, types are covariant in their labels; function types are contravariant in input type and pc types, and covariant in output type. The well-formedness condition $pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)$ of function signatures is enforced in T-SIGNATURE.

The typing judgment has the form $\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau$, where Γ maps variables to IFC types, pc tracks the current control-flow context, p is the principal on which the expression e is being typed, pc_{top}^{self} is the top pc of the current function, e is the expression being typed and τ is the type of the expression. The judgement is implicitly annotated with a program $P = (F, R)$. Rules for unit, boolean, operations, let, and subsumption are standard. T-FIELD queries the program for the type of the top-level function. T-DOWNGRADE change the label of the type related to a value into another label. T-IF is mostly standard, except it is possible for e_i to have different

$\langle e, \Sigma, \mathsf{locks}, t\rangle \to_r \langle e, \Sigma, \mathsf{locks}, t\rangle$ $\qquad$ $\langle e, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle e, \Sigma, \mathsf{locks}, t\rangle$

APP-REMOTE-OK

$$\frac{q \notin A \qquad \mathsf{expc}(E\,[q.g_\ell\ [pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee}) \qquad \mathsf{cur}(E) = p \qquad \mathsf{body}(q.g) = \lambda x.e \\ p \sqcup pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee} \qquad \forall \ell \in \mathsf{locks},\ p \sqcup pc_{ex}^{caller} \sqsubseteq q \sqcup \ell \qquad \mathsf{locks}' = \mathsf{locks}; q \qquad t' = t; \mathsf{call}^{pc}(q)}{\langle E\,[q.g_\ell\ [pc]v]\,, \Sigma, \mathsf{locks}, t\rangle \to_r \langle E\,[\{e\{v/x\}\}_q]\,, \Sigma, \mathsf{locks}', t'\rangle}$$

APP-REMOTE-ABORT

$$\frac{q \notin A \qquad \mathsf{expc}(E\,[q.g_\ell\ [pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee}) \qquad \mathsf{cur}(E) = p \\ (p \sqcup pc_{ex}^{caller} \not\sqsubseteq pc_{ex}^{callee} \text{ or } \exists \ell \in \mathsf{locks}\,.\, p \sqcup pc_{ex}^{caller} \not\sqsubseteq q \sqcup \ell)}{\langle E\,[q.g_\ell\ [pc]v]\,, \Sigma, \mathsf{locks}, t\rangle \to_r \langle \mathsf{abort}, \Sigma, \mathsf{locks}, t\rangle}$$

RETURN-POP

$$\frac{q \notin A \qquad \mathsf{locks} = \mathsf{locks}'; q \qquad t' = t; \mathsf{ret}(q)}{\langle \{v\}_{q.g}, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle v, \Sigma, \mathsf{locks}', t'\rangle}$$

APP-REMOTE-CALLEE-ADV

$$\frac{q \in A \qquad \mathsf{body}(q.g) = \lambda x.e \qquad t' = t; \mathsf{call}^{pc}(q)}{\langle E\,[q.g_\ell\ [pc_{ex}^{caller}]v]\,, \Sigma, \mathsf{locks}, t\rangle \to_r \langle E\,[\{e\{v/x\}\}_{q.g}]\,, \Sigma, \mathsf{locks}, t'\rangle}$$

RETURN-CALLEE-ADV

$$\frac{q \in A \qquad t' = t; \mathsf{ret}(q)}{\langle \{v\}_{q.g}, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle v, \Sigma, \mathsf{locks}, t'\rangle}$$

Fig. 7: Run-time check in λ_{tc}^r. These rules replace APP-REMOTE and RETURN from λ_{tc}.

types; the if-statement is typed-checked with respect to the join of the type of both branches, where the join for types $(\tau_1 \sqcup \tau_2)$ is defined as the least upper bound of τ_1 and τ_2 with respect to the subtyping relation $\preccurlyeq$. T-READ and T-WRITE both type check with respect to the type that the principal p and the ref name h maps to within the type heap R. When interacting with the store, it is necessary that the current label of the type τ is no more trusted than the current pc, as untrusted individual should not access or modify trusted information. T-WRAPPER mirrors that of T-SIGNATURE, which requires that type of the expression in the wrapper to respect the function signature.

T-APP enforces the pc_{ex} and pc_{top} policies from §II. First, it ensures both the calling pc and the label of the function pointer ℓ are no less trusted than the external pc of the function. This ensures that the external pc policy is respected and the attacker influence over the function pointer is accounted for. The check $pc_{ex} \sqsubseteq pc_{top} \sqcup p$ requires that the external pc either flows to the top pc or flows to the caller p. In the first case, the function being called will never auto-endorse before it returns; in the second case, this function call is not an attenuated call, so the control flow integrity p does not fall below p in the first place. Finally, the call checks that pc_{top} of the function being called is bounded by the current top pc pc_{top}^{self} of the caller. This check ensures the promise made pc_{top}^{self} makes about the maximum effect of the current function is not violated indirectly through remote calls. We write $\tau_o \sqcup \ell$ as a shorthand for updating the label of τ_o to $\mathsf{lbl}(\tau_o) \sqcup \ell$.

For type safety, we require that all outputs from the heap Σ are well-typed (T-STORE).

Lemma IV.1 (Type Safety of λ_{tc})**.** *Progress:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge e \neq v \implies \exists e',\ \langle e, \Sigma, t\rangle \to \langle e', \Sigma', t'\rangle$$

Preservation:

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge \langle e, \Sigma, t\rangle \to \langle e', \Sigma', t'\rangle \implies P \vdash \Sigma' \wedge \exists \tau',\ \Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau'$$

Proof. See appendix. □

E. The Real World

We define the calculus λ_{tc}^r of the real world, where the honest principals use dynamic IFC checks.

Attackers from λ_{tc}^r still respect the non-security types—they do not use bools as function pointers and vice versa. Typing rules for attackers in λ_{tc}^r are identical to that of λ_{tc} except that all the IFC subtyping judgments are removed. We show the static semantics of ill-typed attackers in the appendix.

Figure 7 defines the run-time checks in the operational semantics $\to_r$ and $\to_{er}$ of λ_{tc}^r, which is identical to $\to$ and $\to_e$ from λ_{tc} respectively except that APP-REMOTE-OK and APP-REMOTE-ABORT and APP-REMOTE-CALLEE-ADV replace APP-REMOTE from λ_{tc}; RETURN-POP and RETURN-CALLEE-ADV replace RETURN. The honest principals also maintain a dynamic lock list to prevent reentrancy attacks [4]. Each honest principal puts itself into the lock list when it is called and removes itself from the lock list when it returns.

All dynamic checks happen at the callee side. When the callee is controlled by the attacker, no dynamic check happens and the call always succeeds (APP-REMOTE-CALLEE-ADV). The lock list remains intact because the attacker ignores all run-time mechanisms.

Otherwise, the callee is honest and performs two checks after receiving pc_{ex}^{caller} from the caller: 1) whether the caller and the callee agree on the external pc of function being called; 2) whether this call violates an existing reentrancy lock. This run-time check uses the auxiliary function expc to model caller's and callee's knowledge of the call. $\mathsf{expc}(E\,[q.g_\ell\ [pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee})$ given by:

$$pc_{ex}^{caller} = \begin{cases} \bot & \mathsf{cur}(E) = p \text{ and } p \in A \\ pc & \mathsf{cur}(E) = p \text{ and } p \notin A \end{cases}$$

$$pc_{ex}^{callee} = pc_{ex} \text{ where } P(q.g) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e)$$

$$w|_{\bar{A}} = \begin{cases} (p.h := ()) & w = (p.h := q.g) \\ \epsilon & w = (p.h := v),\ \mathsf{lbl}(R(p.h)) \in A \\ \epsilon & w = \mathsf{call}^{pc}(q),\ q \in A \\ \epsilon & w = \mathsf{ret}(q),\ q \in A \\ w & \text{otherwise} \end{cases}$$

$$t|_{\bar{A}} = \begin{cases} t' & t = t'; w,\ w|_{\bar{A}} = \epsilon \\ t'; w|_{\bar{A}} & t = t'; w,\ w|_{\bar{A}} \neq \epsilon \\ \epsilon & t = \epsilon \end{cases}$$

$$t \approx_{\bar{A}} t' \iff t|_{\bar{A}} = t'|_{\bar{A}}$$

Fig. 8: Trace Erasure and Equivalence.

We conservatively assume that attackers always try to bypass the dynamic check when calling honest principals. Any attacker who wishes such a call to fail may be simulated by another attacker who simply aborts before the call. So when the caller is controlled by the attacker, it always sends $pc_{ex}^{caller} = \bot$ to the callee so that the call passes the first external pc check. Otherwise, the caller is trusted and she sends the correct calling pc pc as pc_{ex}^{caller}. In cases where the callee is honest, the callee uses pc_{ex} from its own function signature as pc_{ex}^{callee}.

The check $p \sqcup pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee}$ ensures that the caller uses sufficient integrity to call the callee function. Mathematically, this is two checks in one: $p \sqsubseteq pc_{ex}^{callee}$ and $pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee}$. When the caller is honest and does not lie, this check ensures that the calling pc is no less trusted than the external pc of the callee: $p \sqsubseteq pc \sqsubseteq pc_{ex}$. A confused caller would fail this check when it uses untrusted pc to call a high-external-pc function. When the caller is controlled by the attacker, $pc_{ex}^{caller} = \bot \sqsubseteq pc_{ex}^{callee}$ always passes. In such case, we rely on $p \sqsubseteq pc_{ex}^{callee}$, which upper-bound the lie made about pc_{ex}^{caller} by the attacker's own integrity.

The reentrancy lock check is performed entirely at the callee side. When level ℓ is locked, the function called is either one does not endorse control flow ($pc_{ex} \sqsubseteq q$), or the control flow is no less trusted than the lock level to begin with ($pc \sqsubseteq \ell$).

V. Type Confusion Security

Informally, we require that every *observable behavior* with an ill-typed attacker is also possible with a well-typed attacker. Observable behaviors include writes to trusted heap references, calls to trusted principals, returns from trusted principals, and aborts. Figure 8 defines trace erasure $t|_{\bar{A}}$ and that erases all untrusted events from a trace t. Trace equivalence $\approx_{\bar{A}}$ requires that two traces are identical after erasure. Finally, function values are considered indistinguishable as they are opaque.

Theorem V.1 (Type Confusion Security)**.** *Type confusion security holds between λ_{tc} and λ_{tc}^{r} for all well-typed programs.*

$$\vdash P \implies \forall \vdash^{r} \mathcal{A}^{r}\ \exists \vdash \mathcal{A}^{i}\ \forall \vdash \Sigma,$$
$$(P, \Sigma, \mathcal{A}^{r}) \Downarrow^{r} t \wedge (P, \Sigma, \mathcal{A}^{i}) \Downarrow^{i} t' \implies t \approx_{\bar{A}} t'$$

Proof. We provide a sketch for core insights, and a more detailed version can be found in the appendix.

The proof is in three steps. First, we *construct* an attacker $\mathcal{A}^{i}$ from $\mathcal{A}^{r}$. Second, we show that $\mathcal{A}^{i}$ is well-typed. Finally, we prove behavioral equivalence.

Figure 9 formally defines the simulator construction. The construction of $\mathcal{A}^{i}$ from $\mathcal{A}^{r}$ is defined using $\mathcal{T}[\![\cdot]\!]$, which translates $\mathcal{A}^{r} = (A, M^{r}(A))$ into $(A, M^{i}(A))$ where $M^{i}(A) = \mathcal{T}[\![M^{r}(A), F]\!]$. The key idea in this translation is the construction of new, suitably modified functions from F and $M^{r}(A)$, which we store in $M^{i}(A)$. More precisely, $M^{i}(A)$ contains two new variants of each function: one modified by $pc_{top} = \bot$ and the other by $pc_{top} = \top$. As programs are finite (thus mentioning finitely many functions), and $\mathbb{F}$ is infinite, there are always fresh names for these new functions. $\mathcal{N}[\![\cdot]\!]$ is used to map into these fresh names (we make sure to not have name clashes), and $\mathcal{U}[\![\cdot]\!]$ is used to give type signatures to completely new functions. Original type signatures (S-ORIGINAL) are left unchanged, as they are fixed by the (trusted) program P. The function $\mathsf{C}^{\ell}(\cdot)$ is an auxiliary function that converts between the two modified versions of any function.

Well-typedness of $\mathcal{A}^{i}$ is fairly straightforward, and presented in detail in the appendix. A noteworthy insight of this proof is that the freshly created functions belong to the attacker principal p_A (whose integrity is the least trusted label $\top$). By fixing the control flow at $\top$, which cannot be further lowered, these functions are well-typed and immune to reentrancy attacks.

Finally, we prove behavioral equivalence using bisimulation. Here, we sketch out the main cases: By replacing functions with $\lambda x.\mathsf{abort}$, S-AUTOENDORSE-ABORT and S-HIGH solve pc_{ex} and pc_{top} confusion respectively. Furthermore, they are complemented by S-AUTOENDORSE-SUCCESS and S-LOW when there is no confusion. S-ORIGINAL replaces the original attackers of $\mathcal{A}^{r}$. Detailed proofs are presented in the appendix. □

A. Base Type Confusion

In this section, we describe λ_{tc}^{b} to model base type-confusion attacks where attackers do not follow any typing discipline. λ_{tc}^{b} has the same syntax as λ_{tc} and accepts all terms. To ensure that progress holds for this language, we need to define dynamic semantics for ill-typed operations. For exmple, the language should take a step when the guard of an if statement is a function pointer.

We assume the existence of a low-level abstract semantic domain $\mathbb{C}$, and all values are interpretations of codes from this domain. Let the interpretation be a surjection $[\![c]\!]_{\tau}$, which maps each code $c \in \mathbb{C}$ into a value of type τ. For example, in C, the interpretation function can be instantiated by setting the domain to be the bytes and $[\![c]\!]_{\mathsf{bool}} = \mathsf{false}$ only when c is 0, and true otherwise.

Instead of defining semantics over values, we define them over their codes. When a $(v : \tau^{r})$ is used in elimination forms (like IF-T for bools and APP-REMOTE for functions), the value is reinterpreted into the expected type. Formally,

$\boxed{\mathcal{T}[\![M(A),F]\!](p.f) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e)}$

S-ORIGINAL
$$\frac{M(A)(p.f) = (\eta, \lambda x.e)}{\mathcal{T}[\![M(A),F]\!](p.f) = (\eta, \lambda x.(\mathcal{N}^\top[\![p.f]\!]_\top)\ [\top](\mathsf{C}^\top(x) \downarrow \mathcal{U}^\top[\![\tau_i]\!]))}$$

S-HIGH
$$\frac{F(p.f) = (\eta, \lambda x.e) \qquad pc_{ex} \notin A}{\mathcal{T}[\![M(A),F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{abort})}$$

S-AUTOENDORSE-ABORT
$$\frac{F(p.f) = (\eta, \lambda x.e) \qquad pc_{ex} \in A \qquad p \notin A \qquad \ell = \top}{\mathcal{T}[\![M(A),F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{abort})}$$

S-AUTOENDORSE-SUCCESS
$$\frac{F(p.f) = (\eta, \lambda x.e) \qquad pc_{ex} \in A \qquad p \notin A \qquad \ell = \top}{\mathcal{T}[\![M(A),F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{C}^\ell(p.f_\top\ ((\mathsf{C}^\top(x)) \downarrow \tau_i)))}$$

S-LOW
$$\frac{M(A)(p.f) = (\eta, \lambda x.e) \qquad p \in A}{\mathcal{T}[\![M(A),F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{C}^\ell(\mathcal{S}^\ell[\![e]\!]))}$$

$\boxed{\mathcal{U}^\ell[\![\tau]\!] = \tau}$

$$\mathcal{U}^\ell[\![\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o]\!] = \mathcal{U}^\ell[\![\tau_i]\!]_\top \xrightarrow{\top \lhd \ell} \mathcal{U}^\ell[\![\tau_o]\!]_\top \qquad \mathcal{U}^\ell[\![\mathsf{bool}_{\ell'}]\!] = \mathsf{bool}_\top \qquad \mathcal{U}^\ell[\![1]\!] = 1$$

$\boxed{\mathsf{C}^\ell(v) = v}$

$$\mathsf{C}^\ell(x) = x \qquad \mathsf{C}^\ell(()) = () \qquad \mathsf{C}^\ell(b_{\ell'}) = b_\top \qquad \mathsf{C}^\ell(p.f_{\ell'}) = \mathcal{N}^\ell[\![p.f]\!]_\top \qquad \mathsf{C}^\ell(e) = \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{C}^\ell(x)$$

$\boxed{\mathcal{S}^\ell[\![v]\!] = v}$

$$\mathcal{S}^\ell[\![v]\!] = \mathsf{C}^\top(v) \qquad \mathcal{S}^\ell[\![\mathsf{abort}]\!] = \mathsf{abort} \qquad \mathcal{S}^\ell[\![v \downarrow \ell']\!] = \mathcal{S}^\ell[\![v]\!] \qquad \mathcal{S}^\ell[\![v_1 \otimes v_2]\!] = \mathcal{S}^\ell[\![v_1]\!] \otimes \mathcal{S}^\ell[\![v_2]\!]$$

$$\mathcal{S}^\ell[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] = \mathsf{let}\ x = \mathcal{S}^\ell[\![e_1]\!]\ \mathsf{in}\ \mathcal{S}^\ell[\![e_2]\!] \qquad \mathcal{S}^\ell[\![\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] = \mathsf{if}\ \mathcal{S}^\ell[\![v]\!]\ \mathsf{then}\ \mathcal{S}^\ell[\![e_1]\!]\ \mathsf{else}\ \mathcal{S}^\ell[\![e_2]\!]$$

$$\frac{\mathcal{H}^\ell[\![p.h]\!] = p_A.h'}{\mathcal{S}^\ell[\![\mathsf{read}_h]\!] = \mathsf{read}_{h'}} \qquad \frac{\mathcal{H}^\ell[\![p.h]\!] = p_A.h'}{\mathcal{S}^\ell[\![\mathsf{write}_h(v)]\!] = \mathsf{write}_{h'}(\mathcal{S}^\ell[\![v]\!])} \qquad \mathcal{S}^\ell[\![v_1\ v_2]\!] = \mathsf{C}^\ell(v_1)\ [\top]\mathsf{C}^\ell(v_2) \qquad \mathcal{S}^\ell[\![\{e\}_{q.g}]\!] = \{\mathcal{S}^\ell[\![e]\!]\}_{q.g}$$

Fig. 9: Simulator Construction

in each execution, we use the function $[\![v]\!]^{-1}$ to define the low-level code of v. To ensure that well-typed programs of λ^b_{tc} behave the same as those of λ^r_{tc}, $[\![[\![v]\!]^{-1}]\!]_{\tau^r} = v$ should hold for well-typed values. Note that the semantics of λ^b_{tc} are nondeterministic for ill-typed operations, because multiple codes may encode the same value.

Type confusion security holds between λ^r_{tc} and λ^b_{tc} when fixing $[\![v]\!]^{-1}\tau^r$, the source of nondeterminism.

Lemma V.2 (Base Type Confusion Security)**.** *Type confusion security holds between λ^r_{tc} and λ^b_{tc} for all well-typed P.*

$$\vdash P \implies \forall \vdash^b \mathcal{A}^b\ \exists \vdash^r \mathcal{A}^r\ \forall \vdash \Sigma,\\ (P,\Sigma,\mathcal{A}^b) \Downarrow^b t' \wedge (P,\Sigma,\mathcal{A}^r) \Downarrow^r t \implies t \approx_{\bar{A}} t'$$

Proof sketch: the simulator synthesizes a type derivation tree for the program and simply reinterprets all casted values as the correct type.

More importantly, type confusion security composes.

Lemma V.3 (Base Type Confusion Security)**.** *Type confusion security holds between λ^b_{tc} and λ_{tc} for all well-typed P.*

$$\vdash P \implies \forall \vdash^b \mathcal{A}^b\ \exists \vdash \mathcal{A}^i\ \forall \vdash \Sigma,\\ (P,\Sigma,\mathcal{A}^b) \Downarrow^b t' \wedge (P,\Sigma,\mathcal{A}^i) \Downarrow^i t \implies t \approx_{\bar{A}} t'$$

B. Examples Revisited

For better intuition of the construction of the simulator, we revisit the examples from §II. Let the lattice be a two-point lattice $\{U = \top, T = \bot\}$ where $T \sqsubseteq U$. The attacker adv has label U and the compiler has label T. In the syntax of λ_{tc}, the compiler's type is $((\mathsf{bool}_U \xrightarrow{U \lhd U} 1)_U) \xrightarrow{U \lhd T} 1$ (we ignore the code argument for simplicity).

1) The CDA Attack: The billwriter's type is $\mathsf{bool}_U \xrightarrow{T \lhd T} 1$. The attacker is ill-typed because it uses a high-external-pc function pointer as an argument to a function that takes low-external-pc function pointer only. The ill-typed attacker and its simulator are given as follows:

$$\mathsf{adv.main} := \mathsf{Compiler.compile}_U\ [U]\mathsf{billwriter}_U$$
$$\mathcal{S}^T[\![\mathsf{adv.main}]\!] := \mathsf{C}^\mathsf{T}(\mathsf{Compiler.compile}_U)\ [U]\mathsf{C}^\mathsf{T}(\mathsf{billwriter}_U)$$

The simulator translates the attacker's main function by applying $\mathcal{S}^T[\![\cdot]\!]$ to its body where T label means this function is allowed to auto-endorse. Therefore, the translation replaces both the function pointer and the argument by their T-versions. As shown in fig. 9, the body of $\mathsf{C}^\mathsf{T}(\mathsf{Compiler.compile}_U)$ is:

$$\lambda x.\mathsf{Compiler.compile}_U\ [U]\mathsf{C}^\mathsf{U}(x)$$

It simply calls billwriter with the U-version of the argument, which ensures that the argument never auto-endorses. Finally, by fig. 9, the body of $\mathsf{C}^{\mathsf{U}}(\mathsf{billwriter})$ is abort because it is an attacker-controlled function pointer with trusted external pc. This function aborts when being used, simulating the run-time check failure in the real world.

2) The Reentrancy Attack: The type of the function $\mathsf{adv.reenter}$ is $\mathsf{bool}_U \xrightarrow{U \lhd T} 1$. The attacker is ill-typed because it uses a high-top-pc function pointer as an argument to a function that takes low-top-pc function pointer only. The ill-typed attacker, its simulator and the T-version of the compiler function pointer are defined as follows:

$$\mathsf{adv.main} := \mathsf{Compiler.compile}_U\ [U]\mathsf{adv.reenter}_U$$
$$\mathcal{S}^T[\![\mathsf{adv.main}]\!] := \mathsf{C}^{\mathsf{T}}(\mathsf{Compiler.compile}_U)\ [U]\mathsf{C}^{\mathsf{T}}(\mathsf{adv.reenter}_U)$$
$$\mathsf{C}^{\mathsf{T}}(\mathsf{Compiler.compile}_U) := \lambda x.\mathsf{Compiler.compile}_U\ [U]\mathsf{C}^{\mathsf{U}}(x)$$

By fig. 9, the body of $\mathsf{C}^{\mathsf{U}}(\mathsf{adv.reenter})$ is identical to that of adv.reenter except that all auto-endorsing calls are replaced by aborting functions. The behavior of $\mathsf{C}^{\mathsf{U}}(\mathsf{adv.reenter})$ simulates that of adv.reenter correctly the cases where the reentrancy lock is active.

VI. Applications

Our simulation result ensures preservation of all hyperproperties satisfied by well-typed programs are satisfied in the real world too. The following subsections demonstrate the payoff: Real-world (λ^{τ}_{tc}) programs are CDA and reentrancy-secure, simply because well-typed (λ_{tc}) programs are. In addition, we also formalize noninterference preservation between the two settings.

A. Confused Deputy Attacks

As shown from the example in fig. 1, Confused Deputy Attacks (CDAs) are simply violations of external pc policies. That is, a caller's *calling pc* level must not be less trusted than the callee's *external pc*.

Definition VI.1 (CDA call)**.** Let $P = (F, R)$ be a program. A call event is a CDA call when a caller's *calling pc* level is less trusted than the callee's required *external pc*.

Formally, let $F(q.g) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e)$, then:

$$\mathcal{CDA}(\mathsf{call}^{pc}(q.g)) \iff pc \not\sqsubseteq pc_{ex}$$

Definition VI.2 (CDA Security)**.** We define what it means for the tuple $(P, \Downarrow, Atks)$ to be CDA secure, where P is a program, $\Downarrow$ is its semantics, and $Atks$ is the set of attacks under consideration. Informally, we simply require all its traces to be CDA-event free.

Formally, $(P, \Downarrow, Atks)$ is CDA secure if, for all Σ and for all $\mathcal{A} = (A, M(A)) \in Atks$,

$$(P, \Sigma, \mathcal{A}) \Downarrow t \implies \forall\ w \in t|_{\bar{A}}\ \neg\mathcal{CDA}(w)$$

Well-typed programs are CDA-secure by construction.

Lemma VI.3 (Well-typed programs are CDA-secure)**.** *Let $\vdash P$ be a well-typed program, and $AtksIdeal$ be the set of well-typed ideal-world attacks. Then $(P, \Downarrow, AtksWT)$ is CDA-Secure.*

Proof. The pc_{ex} restriction in T-App of the static semantics (fig. 6) enforces this restriction. □

Theorem VI.4 (CDA Security in the Real World)**.** *Let $\vdash P$ be a well-typed program, and $AtksReal$ be the set of real-world attacks. Then $(P, \Downarrow^r, AtksReal)$ is CDA-Secure.*

Proof. Hyperproperty preservation is ensured by theorem V.1, which we specialize to theorem VI.2 and theorem VI.3. □

B. Reentrancy Attack

A reentrancy happens when an auto-endorsing function is called while a trusted level is locked. A formal definition is facilitated by defining the *call stack*: the sequence of principals actively called in a trace.

Definition VI.5 (Call Stack)**.** The call stack of a trace t is given by $\mathsf{stackof}(t)$. Push and pop operations are defined in the obvious way.

$$\mathsf{stackof}(t) = \begin{cases} \epsilon & t = () \\ \mathsf{push}(\mathsf{stackof}(t'), q) & t = t'; \mathsf{call}^{pc}(q) \\ \mathsf{pop}(\mathsf{stackof}(t')) & t = t'; \mathsf{ret}(q) \\ \mathsf{stackof}(t') & t = t'; other \end{cases}$$

Additionally, we say $t \leq t'$ when t is a prefix of t'.

Definition VI.6 (Reentrancy Security)**.** We define what it means for the tuple $(P, \Downarrow, Atks)$ to be reentrancy-secure, where P is a program, $\Downarrow$ is its semantics, and $Atks$ is the set of attacks under consideration.

Informally, we require that no intermediate trusted-function calls made during an execution of an untrusted call by a trusted function.

Formally, we define $(P, \Downarrow, Atks)$ to be reentrancy-secure:

Take Σ, $\mathcal{A} = (A, M(A)) \in Atks$, and $(P, \Sigma, \mathcal{A}) \Downarrow t$. Then consider all $t' \leq t|_{\bar{A}}$ and let $\mathsf{stackof}(t') = p_1, \ldots, p_n$. We require that in all such cases, if $\exists i\ p_i \notin A \wedge p_{i+1} \in A$, then $\forall k > (i+1)\ p_k \in A$.

Similar to CDA security, well-typed programs are reentrancy-secure in both worlds.

Theorem VI.7 (Reentrancy Security)**.** *Let $\vdash P$ be a well-typed program, $AtksIdeal$ be the set of ideal-world attacks, and $AtksReal$ be the set of real-world attacks.*

Then both $(P, \Downarrow, AtksIdeal)$ and $(P, \Downarrow^r, AtksReal)$ are reentrancy-secure.

Proof. The pc_{top} restriction in T-App of the static semantics fig. 6 gets us the ideal-world result. The real-world result follows from hyperproperty preservation (theorem V.1). □

C. Noninterference

Noninterference is a security hyperproperty that ensures trusted information is not influenced by untrusted data and secret information does not influence public data.

Here, we define noninterference for integrity using low-equivalence $\approx_{\bar{A}}$ between Σ pairs, defined in fig. 8.

Definition VI.8 (Noninterference of Integrity)**.** The program-semantics pair $(P, \Downarrow)$ satisfies noninterference when for attackers who control the same set of principals A, running the program with trusted-equivalent initial configurations results in trusted-equivalent traces.

Formally, consider Σ_1, Σ_2, $\mathcal{A}_1^i = (A, M(A)_1)$, and $\mathcal{A}_2^i = (A, M(A)_2)$. We require

$$\begin{aligned}&\Sigma_1 \approx_{\bar{A}} \Sigma_2 \implies \\ &(P, \Sigma_1, \mathcal{A}_1^i) \Downarrow t_1 \wedge (P, \Sigma_2, \mathcal{A}_2^i) \Downarrow t_2 \implies \\ &t_1 \approx_{\bar{A}} t_2\end{aligned}$$

Theorem VI.9 (Noninterference simulation)**.** *If noninterference holds for $(P, \Downarrow)$, then it also holds for $(P, \Downarrow^r)$.*

Proof. This result also follows from hyperproperty preservation ensured by theorem V.1. □

VII. Related Work and Discussion

1) Partial Typing in Open Systems: Riely and Hennessy [22] and Hennessy et al. [23] both aim to provide secure execution in a decentralized system in which not all code is well-typed. Their computation models are based on the π-calculus and differ from ours in three major ways: 1. instead of RPC calls, principals send code blocks to each other; 2. the type system does not track information flow; 3. the primary theoretical result is subject reduction. Lacking RPC calls, their computational model cannot reason about attacker-controlled runtimes, and cannot model standard CDA attacks.

2) CDA and Capability Systems: Rajani et al. [8] formally characterize CDA vulnerabilities and show that capabilities cannot prevent all CDAs. Our work extends their idea in a more expressive setting: 1. instead of first-class references, we have function values, yielding a more expressive language and a stronger attacker model; 2. our real-world semantics does not assume a centralized, trusted runtime. They showed that CDAs are only prevented when *Full Provenance* is enforced, tracking all influences from both control flow and data. This is the same insight as using information flow control for both control flow and data flow in our work. Our major theoretical result is a simulation theorem that generalizes to other kinds of attacks, and CDA security is just an instance of our results.

3) Reentrancy Security: Our approach to reentrancy is inspired by the work by SeRIF [4]. SeRIF is a core calculus that enforces secure reentrancy using a combination of both static and dynamic IFC, and the attacker model is similar to that of our real world λ_{tc}^r. The language supports a wide range of language features that allows users to toggle between static and dynamic IFC for performance. Our real-world dynamic semantics is analogous to a SeRIF program that uses dynamic IFC mechanisms only. While reentrancy security is not the main focus of our work, it is an essential ingredient of our type-confusion security.

4) Gradual Typing: While the problem of connecting a statically typed ideal world and an ill-typed real world is reminiscent of gradual typing [24, 25] where types can be left statically unspecified, our work solves an orthogonal problem. Unlike gradually typed languages, λ_{tc}^r assumes that all trustworthy top-level functions expose correct typing signatures. While prior work has been done on gradually typed IFC languages [26, 27], they focus on different semantic conditions like gradual guarantee [28] and noninterference [2].

5) Real–Ideal Simulation: Our definition of type confusion security is inspired by real–ideal paradigm widely used in the cryptography literature [5]. Patrignani et al. [29] note that real–ideal simulation can be used as a compiler correctness criterion, which requires that the source level simulate the target-level behavior. When interpreted as compiler correctness, our type-confusion security theorem means that a compiler that compiles the ideal world to the real world is a secure compiler.

6) Control Flow Integrity (CFI): Abadi et al. [30] propose Control Flow Integrity (CFI), a security policy that requires all executions follow paths in a pre-determined control flow graph (CFG). Niu and Tan [31] explores Modular CFI (MCFI), a mechanism that enforces CFI in an open system where independently instrumented libraries are linked dynamically.

In fact, our reentrancy mechanism enforces a form of CFI over trusted principals, but our notion of control-flow integrity is stronger, because it forbids adversarial influence on trusted control flow, except through explicit auto-endorsement. This stronger notion is a key ingredient for proving type-confusion security.

7) Smart Contract Security: Our core calculus and security theorems provides a formal basis for the consensus-based [32, 33] design patterns that originated from the smart contract community. For example, the Checks–Effects–Interactions (CEI) pattern says callbacks should be executed after all state changes are made, enforcing secure reentrancy. The "pull over push" pattern recommends withdrawing over sending in transferring funds, which reduces risk of CDA attacks by reducing attenuated calls by the deputy.

VIII. Conclusions

Defining security in decentralized, reactive systems has long been a challenge. In this work, we introduced type confusion security, a formal security definition that tames this complexity through a compositional approach to attacker models. The key insight for enforcing type confusion security is enforcing security against attacks to control flow integrity, which static security mechanisms struggle to protect against. We identify the two well-known attacks to control flow integrity, CDA and reentrancy attacks, and identify a simple static enforcement mechanism for each in the ideal world with well-typed attackers. We also propose dynamic enforcement mechanisms in the real world with ill-typed attackers, and use type confusion security to show that the real world is as secure as the ideal world. Finally, we demonstrate the applicability of our results by applying type confusion security to various security hyperproperties.

References

[1] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2001, also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[2] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11–20.

[3] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988.

[4] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers, "Compositional security for reentrant applications," in *IEEE Symp. on Security and Privacy*, May 2021.

[5] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2001, pp. 136–145.

[6] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *22nd ACM Symp. on Operating System Principles (SOSP)*, Oct. 2009, pp. 321–334.

[7] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 293–308.

[8] V. Rajani, D. Garg, and T. Rezk, "On access control, capabilities, their equivalence, and confused deputy attacks," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, Jun. 2016, pp. 150–163.

[9] Cointelegraph, "Dexible aggregator hacked for $2M via 'selfSwap' function," https://cointelegraph.com/news/dexibleapp-aggregator-hacked-for-2m-via-selfswap-function, 17 Feb. 2023, accessed August 2023.

[10] CoinDesk, "Cross-chain DeFi site poly network hacked; hundreds of millions potentially lost," https://www.coindesk.com/markets/2021/08/10/cross-chain-defi-site-poly-network-hacked-hundreds-of-millions-potentially-lost, 10 Aug. 2021, accessed August 2023.

[11] ——, "Defi protocol LI.FI struck by $11M exploit," https://www.coindesk.com/business/2024/07/16/defi-protocol-lifi-struck-by-8m-exploit/, 16 Jul. 2024, accessed October 2024.

[12] J. B. Dennis and E. C. VanHorn, "Programming semantics for multiprogrammed computations," *Comm. of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[13] L. Cardelli, "A semantics of multiple inheritance," *Information and Computation*, vol. 76, no. 2–3, pp. 138–164, 1988, also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[14] P. Daian, "Analysis of the DAO exploit," https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 18 Jun. 2016, accessed March 2021.

[15] S. Ren, C. Acay, and A. C. Myers, "An algebraic approach to asymmetric delegation and polymorphic label inference," in *European Symposium on Research in Computer Security (ESORICS)*, Sep. 2025.

[16] S. Zdancewic and A. C. Myers, "Secure information flow and CPS," in *10th European Symposium on Programming*, ser. Lecture Notes in Computer Science, D. Sands, Ed., vol. 2028. Springer Berlin Heidelberg, 2001, pp. 46–61.

[17] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE Computer Society, 2019, pp. 256–271.

[18] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *24th ACM Conf. on Computer and Communications Security (CCS)*. ACM, Oct. 2017, pp. 1875–1891.

[19] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style," *Lisp and Symbolic Computation*, vol. 6, no. 3–4, pp. 289–360, Nov. 1993.

[20] E. Kozyri, S. Chong, and A. C. Myers, "Expressing information flow properties," *Foundations and Trends in Privacy and Security*, vol. 3, no. 1, pp. 1–102, 2022.

[21] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[22] J. Riely and M. Hennessy, "Trust and partial typing in open systems of mobile agents," *J. Automated Reasoning*, vol. 31, no. 3, pp. 335–370, 2003.

[23] M. Hennessy, J. Rathke, and N. Yoshida, "SAFEDPI: A language for controlling mobile code," *Acta Informatica*, vol. 42, no. 4, pp. 227–290, 2005.

[24] J. Siek and W. Taha, "Gradual typing for objects," in *21st European Conf. on Object-Oriented Programming*, Jul. 2007, pp. 2–27.

[25] P. Wadler and R. B. Findler, "Well-typed programs can't be blamed," in *European Symposium on Programming*, 2009.

[26] T. Chen and J. G. Siek, "Quest complete: The holy grail of gradual security," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024.

[27] A. Bichhawat, M. McCall, and L. Jia, "Gradual security types and gradual guarantees," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021, pp. 1–16.

[28] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland, "Refined criteria for gradual typing," in *Summit on Advances in Programming Languages*, 2015.

[29] M. Patrignani, R. S. Wahby, and R. Künneman, "Universal composability is secure compilation," arXiv ePrint 1910.08634, 2019.

[30] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353.

[31] B. Niu and G. Tan, "Modular control-flow integrity," *SIGPLAN Not.*, vol. 49, no. 6, p. 577–587, Jun. 2014.

[32] L. Marchesi, L. Pompianu, and R. Tonelli, "Security checklists for Ethereum smart contract development: patterns and best practices," *Blockchain: Research and Applications*, p. 100367, 2025.

[33] F. Volland, "Solidity patterns: A compilation of patterns and best practices for the smart contract programming language solidity," https://github.com/fravoll/solidity-patterns, 2019.

APPENDIX

A. Real World Static Semantics

Figure 10 shows the static semantics of the real world.

B. Type Safety

Lemma A.1 (Substitution)**.**

$$\Gamma, x : \tau'; pc \vdash e : \tau \ \wedge \ \emptyset; pc \vdash v : \tau' \implies \Gamma; pc \vdash e\{v/x\} : \tau$$

Proof. By induction over typing derivation. □

Lemma A.2 (PC Anti-Monotonicity)**.**

$$pc' \sqsubseteq pc \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \implies \Gamma; pc'; p; pc_{top}^{self} \vdash e : \tau$$

Proof. By induction over the typing derivation. □

Lemma A.3 (Weakening)**.**

$$\Gamma = \Gamma_1; \Gamma_2 \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \implies \Gamma_1; pc'; p; pc_{top}^{self} \vdash e : \tau$$

Proof. By induction over the typing derivation. □

Lemma A.4 (Let Congruence)**.**

$$\langle e_1, \Sigma, t\rangle \to \langle e_1', \Sigma', t'\rangle \implies \langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \Sigma, t\rangle \to \langle \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2', \Sigma', t'\rangle$$

They also hold for $\{e\}_{p.f}$.

Proof. By case analysis over the stepping derivation. □

Lemma A.5 (Progress)**.** *The following holds for* λ_{tc}*:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge e \neq v \implies \exists e' \,.\, \langle e, \Sigma, t\rangle \to \langle e', \Sigma', t'\rangle$$

They also hold for λ_{tc}^r *(replace the respective* $\to$ *and* $\to_e$ *by* $\to_r$ *and* $\to_{er}$*).*

Proof. We do induction over typing derivation of e.

- Case T-ABORT $e = \mathsf{abort}$. ABORT applies.
- Case T-OP $e = v_1 \otimes v_2$. By typing rule, e_1 and e_2 are bools. So OP applies.
- Case T-LET $e = \mathsf{let}\ x = e'\ \mathsf{in}\ e$. In the case when $\langle e', \Sigma, t\rangle \to \langle e'', \Sigma', t'\rangle$, the congruence lemma A.4 applies. Otherwise, LET applies.
- Case T-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By typing rule, v is a bool. So IF-TRUE or IF-FALSE applies.
- Case T-DOWNGRADE $e = b_{\ell'} \downarrow \ell$. DOWNGRADE applies.
- Case T-READ $e = \mathsf{read}_h$. READ applies.
- Case T-WRITE $e = \mathsf{write}_h(v)$. WRITE applies.
- Case T-APP $e = v_1\ [pc]v_2$. By typing rule, v_1 has function type. So APP-REMOTE applies.
- Case T-WRAPPER $e = \{e'\}_{p.f}$. In the case when $\langle e', \Sigma, t\rangle \to \langle e'', \Sigma', t'\rangle$, the congruence lemma A.4 applies. Otherwise, RETURN and the induction hypothesis applies.
- Case T-SUB $e = e$. The induction hypothesis applies.

To show progress for λ_{tc}^r, we need to show that one of APP-REMOTE-OK, APP-REMOTE-ABORT and APP-REMOTE-CALLEE-ADV applies when APP-REMOTE applies in λ_{tc}, and one of RETURN-POP and RETURN-CALLEE-ADV applies when RETURN applies. This follows from the premises of the rules, which exhaust all cases. □

Lemma A.6 (E-Preservation)**.** *The following holds for* λ_{tc}*:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \ \wedge \langle e, \Sigma, t\rangle \to_e \langle e', \Sigma', t'\rangle \implies \vdash \Sigma' \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau$$

They also hold for λ_{tc}^r *(replace the respective* $\to$ *and* $\to_e$ *by* $\to_r$ *and* $\to_{er}$*).*

Proof. We do induction over typing derivation of e. In all cases except T-WRITE, taking a step does not change the store, so $\vdash \Sigma'$ holds by $\vdash \Sigma$. Moreover, T-WRITE and T-READ are immediately discharged because $\mathsf{write}_h(v)$ and read_h are not in the stepping relation $\to_e$.

- Case T-OP $e = v_1 \otimes v_2$. Then it must be that $v_i = b_{l_i}$ by the interpretation function, and $\Gamma; pc \vdash b_{\ell_i'} : \mathsf{bool}_{lbl_i}$, which implies that $\mathsf{bool}_{\ell_1' \sqcup \ell_2'} \preccurlyeq \mathsf{bool}_{\ell_1 \sqcup \ell_2}$. Preservation holds by T-SUB.
- Case T-LET $e = \mathsf{let}\ x = e'\ \mathsf{in}\ e$. LET applies and $e' = e\{v/x\}$. Preservation holds by theorem A.1.
- Case T-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By typing derivation, $\tau = \tau_1 \sqcup \tau_2$ where e_i have type τ_i. IF-TRUE or IF-FALSE apply and $e' = e_i$ for some i. Let $\Gamma; pc \vdash b : b_\ell$, then by typing derivation, we know $\Gamma; pc \sqcup \ell \vdash e_i : \tau_i$. By theorem A.2, $\Gamma; pc \vdash e_i : \tau_i$ holds. Rule IF-T or IF-F apply.
- Case T-DOWNGRADE $e = b_{\ell'} \downarrow \ell$. DOWNGRADE applies and the bool value remains well-typed.
- Case T-READ $e = \mathsf{read}_h$. Vacuously true.
- Case T-WRITE $e = \mathsf{write}_h(v)$. Vacuously true.
- Case T-APP $e = f\ [pc]v$. By the typing derivation, $e = p.f\ [pc]v$, $\Gamma; pc; p; pc_{top}^{self} \vdash f : (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell$, and $\Gamma; pc; p; pc_{top}^{self} \vdash v : \tau$. By the stepping derivation, $\mathsf{body}(p.f) = e_{p.f}$, and $e' = \{e_{p.f}\{v/x\}\}_{q.g}$ To prove the goal, we do induction on the first typing judgment of the following more general lemma:

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash p.f : (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell \ \wedge \Gamma; pc; p; pc_{top}^{self} \vdash v : \tau \wedge \tau \preccurlyeq \tau_i \wedge \mathsf{body}(p.f) = e_{p.f} \implies \Gamma; pc; p; pc_{top}^{self} \vdash \{e_{p.f}\{v/x\}\}_{q.g} : \tau_o$$

  The proof only involves two cases

  - Case T-FIELD. Case analysis on $q \in A$. By T-SIGNATURE, in either case both the normal function and attacker's function both type-checked with the same type (i.e. $x : \tau_i; p; (p, pc_{top}) \vdash e_{p.f} : \tau_o$). Preservation follows by T-WRAPPER and theorem A.1, theorem A.3, and the fact that pc is irrelevant in type-checking values.

Base Type $\tau^r ::= 1 \mid 0 \mid \mathsf{bool} \mid \tau^r \to \tau^r$

$B(1) = 1 \qquad B(0) = 0 \qquad B(\mathsf{bool}_\ell) = \mathsf{bool} \qquad B(\eta_\ell) = B(\eta) \qquad B(\eta_\ell \xrightarrow{pc_{ex} \triangleleft pc_{top}} \eta'_{\ell'}) = B(\eta) \to B(\eta')$

$\boxed{\vdash P}$

A-SIGNATURE
$$\frac{p \vdash (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_p \qquad x : B(\tau_i); p \vdash^r e : B(\tau_o)}{\vdash^r p.f : \tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o := \lambda x.e}$$

$\boxed{\Gamma^r; p \vdash^r e : \tau^r}$

A-UNIT
$$\frac{}{\Gamma^r; p \vdash^r () : 1}$$

A-BOOL
$$\frac{}{\Gamma^r; p \vdash^r b_\ell : \mathsf{bool}}$$

A-FIELD
$$\frac{q.g : \eta := e \in P}{\Gamma^r; p \vdash^r q.f_\ell : B(\eta)}$$

A-VAR
$$\frac{}{\Gamma^r, x : \tau^r; p \vdash^r x : \tau^r}$$

A-ABORT
$$\frac{}{\Gamma^r; p \vdash^r \mathsf{abort} : 1}$$

A-OP
$$\frac{\Gamma^r; p \vdash^r v_i : \mathsf{bool}}{\Gamma^r; p \vdash^r v_1 \otimes v_2 : \mathsf{bool}}$$

A-LET
$$\frac{\Gamma^r; p \vdash^r e_1 : \tau_1^r \qquad \Gamma^r, x : \tau_1^r; p \vdash^r e_2 : \tau_2^r}{\Gamma^r; p \vdash^r \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2^r}$$

A-IF
$$\frac{\Gamma^r; p \vdash^r v : \mathsf{bool} \qquad \Gamma^r; p \vdash^r e_i : \tau_i^r}{\Gamma^r; p \vdash^r \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1^r \sqcup \tau_2^r}$$

A-READ
$$\frac{}{\Gamma^r; p \vdash^r \mathsf{read}_h : B(\tau)}$$

A-WRITE
$$\frac{\Gamma^r; p \vdash^r v : \tau^r \qquad B(\tau) \preccurlyeq \tau^r}{\Gamma^r; p \vdash^r \mathsf{write}_h(v) : 1}$$

A-DOWNGRADE
$$\frac{}{\Gamma^r; p \vdash^r v \downarrow \ell : \mathsf{bool}}$$

A-APP
$$\frac{\Gamma^r; p \vdash^r v_1 : \tau_i^r \to \tau_o^r \qquad \Gamma^r; p \vdash^r v_2 : \tau_i^r}{\Gamma^r; p \vdash^r v_1\ [pc] v_2 : \tau_o^r}$$

A-WRAPPER
$$\frac{\Gamma^r; p \vdash^r e : \tau^r}{\Gamma^r; p \vdash^r \{e\}_{q.g} : \tau^r}$$

Fig. 10: Static semantics of λ^r_{tc}.

  - Case T-SUB. By typing derivation, $\Gamma; pc; p; pc_{top}^{self} \vdash p.f : \tau'$ and $\tau' \preccurlyeq (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell$, which implies $\tau' = (\tau'_i \xrightarrow{pc'_{ex} \triangleleft pc'_{top}} \tau'_o)_{\ell'}$, and $\tau'_o \preccurlyeq type_o$. The proof follows by induction hypothesis

  The proof of this case follows as an instance of the aforementioned lemma, with $\tau_o \preccurlyeq \tau_o$.
- Case T-WRAPPER $e = \{v\}_{p.f}$. RETURN applies and $e = v$. Preservation holds by typing derivation and the fact that pc is irrelevant in type-checking values.

To show preservation of $\to_{er}$ for λ^r_{tc}, we need to show preservation for APP-REMOTE-CALLEE-ADV, RETURN-POP, and RETURN-CALLEE-ADV, which runs a similar argument as APP-REMOTE and RETURN.

□

Lemma A.7 (Preservation-Eval)**.** *The following holds for λ_{tc}:*

$$\begin{aligned}&\Gamma; pc; p; pc_{top}^{self} \vdash E\,[e] : \tau \\ &\wedge (\forall\ \Gamma'\ pc'\ e'\ \tau', \Gamma'; pc'; p; pc_{top}^{self} \vdash e : \tau' \\ &\quad \implies \Gamma'; pc'; p; pc_{top}^{self} \vdash e' : \tau') \\ &\implies \Gamma; pc; p; pc_{top}^{self} \vdash E\,[e'] : \tau\end{aligned}$$

Proof. Induction over the evaluation context E

- Case $[\cdot]$. The proof follows by the second assumption.
- Case let $x = E$ in e. By the typing derivation, we get $\Gamma; pc; p; pc_{top}^{self} \vdash E[e] : \tau'$, and $\tau'; \Gamma; pc; p; pc_{top}^{self} \vdash E[e] : \tau''$, and $\tau'' \preccurlyeq \tau'$. Preservation-Eval holds by T-SUB, T-LET, and the induction hypothesis.
- Case $\{E\}_p$. By the typing derivation, we get $F(q.g) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e')$, $\tau'_o \preccurlyeq \tau_o$, and $x : \tau_i; q; q; pc_{top} \vdash e' : \tau'_o$. Preservation-Eval holds by T-SUB T-WRAPPER, and the induction hypothesis.

□

Lemma A.8 (Preservation)**.** *The following holds for λ_{tc}:*

$$\begin{aligned}&\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \\ &\quad \wedge \langle e, \Sigma, t\rangle \to \langle e', \Sigma', t'\rangle \\ &\quad \implies P \vdash \Sigma' \wedge \exists \tau' \,.\, \Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau'\end{aligned}$$

They also hold for λ^r_{tc} (replace the respective $\to$ and $\to_e$ by $\to_r$ and $\to_{er}$).

Proof. We do induction over the stepping derivation. In all cases except T-WRITE, taking a step does not change the store, so $\vdash \Sigma'$ holds by $\vdash \Sigma$.

- Case EVAL-EXPR. Preservation follows by theorem A.7 and theorem A.6.
- Case ABORT. Immediately holds as abort can always type check.

$e \approx e'$

$$\frac{\ell \in A \qquad \mathsf{cur}(E) \in A \qquad \mathsf{C}^\top(v) = \mathsf{C}^\top(v')}{E\,[q.g_\ell] \approx E\,[q'.g'_{\ell'}]}$$

$$\frac{\ell \notin A \qquad \mathsf{cur}(E) \in A \qquad q.g_\ell = q'.g'_{\ell'}}{E\,[q.g_\ell] \approx E\,[q'.g'_{\ell'}]}$$

$$\frac{\mathsf{cur}(E) \in A \qquad v'_i = \mathsf{C}^\ell(v_i)}{E\,[v_1\ v_2] \approx E\,[v'_1\ v'_2]}$$

$$\frac{\text{other redex cases} \qquad \mathsf{C}^\ell(e) = \mathsf{C}^\ell(e')}{v \approx v'}$$

$$\frac{E\,[e_i] \approx E\,[e'_i]}{E\,[\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2] \approx E\,[\mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2]}$$

$$\frac{q \notin A \qquad \mathsf{cur}(E) \in A \qquad q.g = q'.g' \qquad E\,[e] \approx E\,[e']}{E\,[\{e\}_{q.g}] \approx E\,[\{e'\}_{q'.g'}]}$$

$$\begin{aligned}
&\Sigma \approx \Sigma' \iff \forall p, h \\
&p \notin A \implies \Sigma(p.h) = \Sigma'(p.h) \wedge \\
&p \in A \implies \Sigma(\mathcal{H}^\top[\![p.h]\!]) = \Sigma'(\mathcal{H}^\top[\![p.h]\!])
\end{aligned}$$

$$w|_C = \begin{cases} (p.h := \mathsf{C}^\top(v)) & w = (p.h := v) \\ w & \text{otherwise} \end{cases}$$

$$t|_C = \begin{cases} t' & t = t'; w,\ w|_C = \epsilon \\ t'; w|_C & t = t'; w,\ w|_C \neq \epsilon \\ \epsilon & t = \epsilon \end{cases}$$

$$t \approx t' \iff t|_C = t'|_C$$

Fig. 11: A stronger trace equivalence.

- Case READ. By theorem A.7, we want to show that $\Gamma'; pc'; p; pc^{self}_{top} \vdash \mathsf{write}_h(v) : \tau' \implies \Gamma'; pc'; p; pc^{self}_{top} \vdash () : \tau'$, where $\Sigma(p.h) = v$, and $R(p.h) = \tau'$. By T-STORE, read_h is type-checked by the corresponding type in R, which proves the claim.
- Case WRITE. By theorem A.7, we want to show that $\Gamma'; pc'; p; pc^{self}_{top} \vdash \mathsf{write}_h(v) : 1 \implies \Gamma'; pc'; p; pc^{self}_{top} \vdash () : 1$. The claim holds due to T-WRITE and T-UNIT.

To show preservation for λ^r_{tc}, we need to show preservation for all APP-REMOTE-OK and APP-REMOTE-ABORT. They run the same argument as APP-REMOTE above. □

C. Proof of Type Confusion

We prove type confusion using the following unwinding lemmas for $\rightarrow_{er}$ and $\rightarrow_r$ respectively. We use a stronger trace equivalence relation $\approx$ defined in fig. 11. Expressions

Theorem A.9 (One Step Type Confusion Security)**.** *Type confusion security between λ_{tc} and λ^r_{tc} for all well-typed programs holds for $\rightarrow_r$.*

$$\begin{aligned}
&\vdash P \wedge P \vDash \Sigma_{\mathcal{A}} \wedge P \vDash \Sigma_{\mathcal{A}^i}, \wedge \Sigma_{\mathcal{A}} \approx \Sigma_{\mathcal{A}^i} \wedge \\
&\Gamma; pc; p; \ell \vdash e_{\mathcal{A}} : \tau_{\mathcal{A}}\ \wedge e_C \approx e_D\ \wedge t_C \approx t_D\ \wedge \\
&\langle e_C, \Sigma_C, \mathsf{locks}, t_C\rangle \rightarrow_r \langle e_C, \Sigma'_C, \mathsf{locks}', t'_C\rangle \implies \\
&\exists e'_D\ \Sigma'_D\ t'_D, \langle e_D, \Sigma_D, t_D\rangle \rightarrow^* \langle e'_D, \Sigma'_D, t'_D\rangle \wedge e'_C \approx e'_D \wedge \\
&\Sigma'_C \approx \Sigma'_D \wedge t'_C \approx t'_D
\end{aligned}$$

Proof. We do case analysis over the stepping relation of the real world $\rightarrow_{er}$.

- Case EVAL-EXPR. By the typing derivation, $e_C = E\,[e]$, $e'_C = E\,[e']$, and $\langle e, \Sigma_C, t_C\rangle \rightarrow_{er} \langle e', \Sigma'_C, t'_C\rangle$. We want to find e'_D such that $\langle \mathcal{S}^\cdot[\![E[e]]\!]\Sigma_D, t_D\rangle \rightarrow e'_D$. We do induction over the stepping relation of the real world $\rightarrow_{er}$. All except WRITE have $\Sigma'_D = \Sigma_D \approx \Sigma_C = \Sigma'_C$.
  - Case OP. By the typing derivation, $e'_C = v[\![\otimes]\!]v'$, which implies that $v = b_l$, $v = b'_{l'}$, and, as a consequence, $e_D = b_l \otimes b_{l'}$. Similarly, by typing derivation, $t_C \approx t_D$ and $\Sigma'_C = \Sigma_C \approx \Sigma_D$ (following a similar argument as above that most of the $\rightarrow_{er}$ rules do not change Σ. By theorem A.11 and theorem A.12, the case holds by EVAL-EXPR and OP.
  - Case IF-T and IF-F. Similar as above.
  - Case DOWNGRADE. Note that $\mathcal{S}^=[\![e_C]\!]\mathcal{S}^{l''}[\![b_{\ell'}\ell\downarrow]\!] = b_\top = e_D$, and Σ_C and t_C do not change. The case holds by taking zero step.
  - Case LET. We first note a fact that translation ($\mathcal{S}^\ell[\![e]\!]$) and substitution ($e\{v/x\}$) are commuting:

    $$\mathcal{S}^\ell[\![e\{v/x\}]\!] = \mathcal{S}^\ell[\![e]\!]\{\mathcal{S}^\ell[\![v]\!]/x\}$$

    The proof of this fact involves induction over the expression e, and the fact that translation is idempotent.

    For the case, by typing derivation, $\mathcal{S}^\ell[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] = \mathsf{let}\ x = \mathcal{S}^\ell[\![v]\!]\ \mathsf{in}\ \mathcal{S}^\ell[\![e]\!]$. By the above lemma, $e'_D = E[\mathcal{S}^\ell[\![e\{v/x\}]\!]]$, which proves the claim using theorem A.11 and theorem A.12.
  - Case RETURN-POP. By typing derivation, $q \notin A$, $t' = t_C; \mathsf{ret}(q)$, $e'_C = v$, and $e_C = \{v\}_{q.g}$. By definition, $\mathcal{S}^\ell[\![\{v\}_{q.g}]\!] = \{\mathcal{S}^\ell[\![v]\!]\}_{q.g}$. Because $\mathsf{C}^\top(v)$ is also value. Let $e'_D = E[\mathcal{S}^\ell[\![v]\!]]$. By theorem A.11 and theorem A.12, the case is proven by EVAL-EXPR and RETURN. RETURN-CALLEE-ADV follows the same proof structure.
- Case ABORT. This case holds by taking one step using theorem A.11, theorem A.12.
- Case READ. By typing derivation, $\mathsf{cur}(E) = p$, $\Sigma(p.h) = v$, $e_C = E\,[\mathsf{read}_h]$, and $e'_C = E\,[v]$. By assumption, e_D will equal to some $E'[\mathsf{read}_h]$, where $\mathcal{H}^\ell[\![p.h]\!] = p_A.h'$. Let $e'_D = E'[v']$, where $\Sigma'_D(p, h) = v'$. The case holds by the assumption that $\Sigma_C \approx \Sigma_D$. $t'_C = t_C \approx t_D = t'_D$, and $e'_C \approx e'_D$ by theorem A.11 and theorem A.12.
- Case WRITE. By typing derivation, $\mathsf{cur}(E) = p$, $\Sigma' = \Sigma\,[p.h \mapsto v]$, $t' = t; (p.h := v)$, $e_C = E\,[\mathsf{write}_h(v)]$, and $e'_C = E\,[()]$. By assumption, e_D will equal to some

$E'[\mathsf{write}_h(v)]$, where $\mathcal{H}^\ell[\![p.h]\!] = p_A.h'$. Let $e'_D = E'[()]$, where $\Sigma'_D = \Sigma_D\,[p_A.h' \mapsto v]$. Then $\Sigma'_C \approx \Sigma'_D$. Also, $t'_C \approx t'_D$ by definition, and $e'_C \approx e'_D$ by theorem A.11 and theorem A.12.

- Case APP-REMOTE-CALLEE-ADV
  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.
  $e'_C = \{e\{v_2/x\}\}_{q.g}$;e is the body of $q.g$. Let
  $e_D = v'_1\ [\top]v'_2$. This function takes one step to
  $e'_D = \{e'\{v'_2/x\}\}_{q'.g'}$;e' is body of $v'_1 = q'.g'_{\ell'_1}$.
  It remains to show $\{e\{v_2/x\}\}_{q.g} \approx \{e'\{v'_2/x\}\}_{q'.g'}$.
  Because $q \in A$ holds by premise of APP-REMOTE-CALLEE-ADV, by definition of $\approx$ and commutivity of $\approx$ and substitution we only need to show $\mathsf{C}^\top(v_i) = \mathsf{C}^\top(v'_i)$. This holds in general by definition of $\approx$.
- Case APP-REMOTE-OK.
  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.
  $e'_C = \{e\{v_2/x\}\}_{q.g}$;e is the body of $q.g$. Let
  $e_D = v'_1\ [\top]v'_2$. This function takes one step to
  $e'_D = \{e'\{v'_2/x\}\}_{q'.g'}$;e' is body of $v'_1 = q'.g'_{\ell'_1}$. Let the signature of v_1 be $\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o$.
  It remains to show $\{e\{v_2/x\}\}_{q.g} \approx \{e'\{v'_2/x\}\}_{q'.g'}$.
  Because $q \notin A$ holds by premise of APP-REMOTE-OK, by definition of $\approx$ we need to show:
  (1)$q'.g' = \mathsf{C}^\ell(q.g)$, (2) $e \approx e'$ and (3) $v_2 \approx v'_2$.
  Case on if $\mathsf{cur}(E) = p \in A$ or not.
  - $\mathsf{cur}(E) \in A$. Note that by premise of APP-REMOTE-OK, $\ell = \top$, so S-AUTOENDORSE-SUCCESS applies for e'. By definition of $\approx$ and S-AUTOENDORSE-SUCCESS , we have $v'_i = \mathsf{C}^\ell(v_i)$, which proves (1) and (2).
    Since $\mathsf{cur}(E) \in A$, $\ell_2 \in A$ by typing derivation of the body of $q.g$. (3) holds by definition of $\approx$.
  - $\mathsf{cur}(E) \notin A$. We first case on ℓ_1:
    * $\ell_1 \notin A$. In this case $v_1 = v'_1$ and S-ORIGINAL applies. (1) and (2) hold. (3) holds by another casing over the label of v_2.
    * $\ell_1 \in A$. By definition of $\approx$ and S-HIGH, we have $\mathsf{C}^\top(v_i) = \mathsf{C}^\top(v'_i)$. Then by definition of $\approx$, (1), (2), and (3) all hold.
- Case APP-REMOTE-ABORT
  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.
  $e'_C = \mathsf{abort}$. Let
  $e_D = v'_1\ [\top]v'_2$. This function takes one step to
  $e'_D = \{e'\{v'_2/x\}\}_{q'.g'}$;e' is body of $v'_1 = q'.g'_{\ell'_1}$. We need to show that $e' = \mathsf{abort}$. Let the signature of v_1 be $\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o$.
  By premise of APP-REMOTE-ABORT, either $pc_{ex} \notin A$ or $\ell = \top$ holds. In the first case e' takes the body of S-HIGH and in the second case the body takes of S-AUTOENDORSE-ABORT. It aborts in both cases.

□

Theorem A.10 (Type Confusion Security)**.** *Type confusion security holds between λ_{tc} and λ^τ_{tc} for all well-typed programs.*

$$\begin{aligned}&\vdash P \wedge P \vDash \Sigma_{\mathcal{A}} \wedge P \vDash \Sigma_{\mathcal{A}^i}, \wedge\Sigma_{\mathcal{A}} \approx \Sigma_{\mathcal{A}^i} \wedge\\ &\Gamma; pc; p; \ell \vdash e_{\mathcal{A}} : \tau_{\mathcal{A}}\ \wedge e_C \approx e_D\ \wedge t_C \approx t_D\ \wedge\\ &\langle e_C, \Sigma_C, \mathsf{locks}, t_C\rangle \to_r \langle e_C, \Sigma'_C, \mathsf{locks}', t'_C\rangle\ \implies\\ &\exists e'_D\ \Sigma'_D\ t'_D, \langle e_D, \Sigma_D, t_D\rangle \to^* \langle e'_D, \Sigma'_D, t'_D\rangle \wedge e'_C \approx e'_D \wedge\\ &\Sigma'_C \approx \Sigma'_D \wedge t'_C \approx t'_D\end{aligned}$$

Proof. Induction on reflexive, transitive closure of the step. The claim vacuously holds for the reflexive case. For inductive case, we apply theorem A.9 and the induction hypothesis. □

Lemma A.11 (Translation of Evaluation Context)**.** *The following holds for translation of expression and evaluation context:*

$$\forall e\ E, \exists E', \mathcal{S}^\ell[\![E\,[e]]\!] = E'[\mathcal{S}^\ell[\![e]\!]]$$

Proof. Induction on E. We present here the LET case.

- Case $E = \mathsf{let}\ x = E'\ \mathsf{in}\ e'$. Therefore, $\mathcal{S}^\ell[\![E[e]]\!] = \mathsf{let}\ x = \mathcal{S}^\ell[\![E'[e]]\!]\ \mathsf{in}\ e'$. By induction hypothesis, there exists E'' such that $\mathcal{S}^\ell[\![E''[e]]\!] = E''[\mathcal{S}^\ell[\![e]\!]]$. Let $E' = \mathsf{let}\ x = E''\ \mathsf{in}\ e'$ proves the claim.

□

Lemma A.12 (Uniqueness of Translation of Evaluation Context)**.** *The following holds for translation of expression and evaluation context:*

$$\begin{aligned}&\forall e_1\ e_2\ E\ E_1\ E_2,\\ &\mathcal{S}^\ell[\![E[e_1]]\!] = E_1[\mathcal{S}^\ell[\![e_1]\!]] \wedge\\ &\mathcal{S}^\ell[\![E[e_2]]\!] = E_2[\mathcal{S}^\ell[\![e_2]\!]]\\ &\implies E_1 = E_2\end{aligned}$$

Proof. Induction on E. This holds true because $\mathcal{S}^\ell[\![\cdot]\!]$ is a function. □

Lemma A.13 (Well-typed Translation)**.** *Let $\mathsf{C}^\ell(p.f) = p_A.g^\ell$, $,F(p.f) = (\eta, \lambda x.e)$ and $,F(p_A.g^\ell) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.e')$. Let $\eta = \tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o$. Then:*

$$x : \tau; p \vdash^r e : \tau\ \implies\ x : \mathcal{U}^\ell[\![\tau_i]\!]; \top; \top; \ell \vdash e' : \mathcal{U}^\ell[\![\tau]\!]$$

Proof. We do induction over the translation $\mathcal{S}[\![\cdot]\!]$.

- Case B-FIELD-DEF Let $\mathcal{U}^\ell[\![\tau_1]\!] = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top} \sqcup \ell} \tau_o)_\top$ and $\mathcal{U}^\ell[\![\tau_2]\!] = \tau$.
  We need to show the following: $pc \sqcup \ell \sqsubseteq pc_{ex}$, $pc_{ex} \sqsubseteq pc_{top} \sqcup \ell \sqcup p$, $pc^{self}_{top} \sqsubseteq pc_{top}$ and $\tau \preccurlyeq \tau_i$. Note that $pc_{ex} = \ell = pc = p = pc_{top} = \top$, $\tau = \mathcal{U}^\top[\![\tau_2]\!] = \tau_i$ So all four conditions hold.
- Case B-UNIT, B-BOOL, B-FIELD, B-VAR hold by typing derivation of λ_{tc}.
- Case B-ABORT.
- Case B-OP, B-LET, B-ENDORSE. Holds by inductive hypothesis and typing derivation of λ_{tc}.
- Case B-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Since the guard is downgraded to p_A the branches also type check under the pc p_A. Holds by inductive hypothesis and typing derivation of λ_{tc}.

- Case B-READ. To show this is well-typed, we need to show $pc \sqsubseteq \mathsf{lbl}(\tau)$. This follows from the definition of $\mathsf{C}^\ell(p.h)$ which ensures $\mathsf{lbl}(\tau') = \top = pc$.
- B-WRITE. To show $\Pi \vdash \mathsf{write}_{\mathsf{C}^\ell(p.h)}(v) : \tau$, we need to show $\tau' \preccurlyeq \tau$ and $pc \sqsubseteq \mathsf{lbl}(\tau)$. This holds by definition of $\mathsf{C}^\ell(p.h)$ and translation of values.
- Case B-APP By theorem A.13, let $\mathcal{U}^\ell[\![\tau_1]\!] = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top} \sqcup \ell} \tau_o)_\top$ and $\mathcal{U}^\ell[\![\tau_2]\!] = \tau$.
  We need to show the following: $pc \sqcup \ell \sqsubseteq pc_{ex}$, $pc_{ex} \sqsubseteq pc_{top} \sqcup \ell \sqcup p$, $pc_{top}^{self} \sqsubseteq pc_{top}$ and $\tau \preccurlyeq \tau_i$. Note that $pc_{ex} = \ell = pc = p = \top$, $pc_{top} = \ell' \sqcup \ell$, $pc_{top}^{self} = \ell$, $\tau = \mathcal{U}^\ell[\![\tau_2]\!] = \tau_i$ So all four conditions hold.
- Case B-WRAPPER

Then we show that all attacker functions are well-typed. This requires shoing $\tau \preccurlyeq \tau_o$. Case 1 AT-SIGNATURE-TOP: holds by definition of $\mathcal{U}^\ell[\![\cdot]\!]$. Case 2 AT-SIGNATURE-EX: Same. □

Lemma A.14 (Simulation Relation)**.**

$$t \approx t' \implies t \approx_{\bar{A}} t'$$

Proof. By induction over the definition of $\approx$. □

Lemma A.15 (Reentrancy Security, Real World)**.** *At any point of the execution, if a trusted function calls an untrusted function, then all functions called after that are also untrusted.*

$$\langle \{e\}_{p_A.\mathsf{main}}, \Sigma, () \rangle \to^{r*} \langle e', \Sigma, t \rangle \wedge$$
$$\mathsf{stack}(t) = p_1.f_1, \ldots, p_n.f_n \implies$$
$$\exists i < j \in [1,n] \,.\, p_i \notin A \wedge p_j \in A \implies \forall k \in [j,n] \,.\, p_k \in A$$

Proof. By theorem A.16 and the definition of $\to^r$. □

Lemma A.16 (Lock)**.**

$$\langle \{e\}_{p_A.\mathsf{main}}, \Sigma, \mathsf{locks}, () \rangle \to^{r*} \langle e', \Sigma, \mathsf{locks}', t \rangle$$
$$\wedge \ \mathsf{stack}(e') = p_1.f_1, \ldots, p_n.f_n \implies \mathsf{locks}' \neq \emptyset$$

Proof. Note that the lock list only contains labels of trusted functions. The lemma hold by induction over the operational semantics. □

APPENDIX