Securing Cryptographic Software via Typed Assembly Language

ABSTRACT

Authors of cryptographic software are well aware that their code should not leak secrets through its timing behavior, and, until 2018, they believed that following industry-standard *constant-time* coding guidelines was sufficient. However, the revelation of the Spectre family of speculative execution attacks injected new complexities.

To block speculative attacks, prior work has proposed annotating the program's source code to mark secret data, with hardware using this information to decide when to speculate (i.e., when only public values are involved) or not (when secrets are in play). While these solutions are able to track secret information stored on the heap, they suffer from limitations that prevent them from correctly tracking secrets on the stack, at a cost in performance.

This paper introduces *SecSep*, a transformation framework that rewrites assembly programs so that they partition secret and public data on the stack. By moving from the source-code level to assembly rewriting, *SecSep* is able to address limitations of prior work. The key challenge in performing this assembly rewriting stems from the loss of semantic information through the lengthy compilation process. The key innovation of our methodology is a new variant of typed assembly language (TAL), *Octal*, which allows us to address this challenge. Assembly rewriting is driven by compile-time inference within *Octal*. We apply our technique to cryptographic programs and demonstrate that it enables secure speculation efficiently, incurring a low average overhead of 1.2%.

CCS CONCEPTS

• **Security and privacy**;

KEYWORDS

Side-channel attacks and mitigation, Information-flow security

ACM Reference Format:

. 2025. Securing Cryptographic Software via Typed Assembly Language. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25).* ACM, New York, NY, USA, 34 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cryptographic software has strong security requirements and is often strengthened to prevent information leakage through timing side channels by adhering to the *constant-time coding discipline*, which requires programs to avoid using secret-dependent values as branch conditions or memory addresses.

CCS '25, Oct 13–17, 2025, Taipei, Taiwan

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

However, recent speculative-execution attacks, notably various Spectre attacks [34–36, 39, 50], have invalidated the security guarantees offered by constant-time programming. Modern processors employ aggressive speculative-execution mechanisms that predict upcoming instructions to be executed and roll back architectural state if the prediction is later found to be incorrect. While offering significant performance benefits, such speculative-execution mechanisms introduce a large attack surface, enabling attackers to trigger a program to execute unintended instructions speculatively to access secrets and transmit them via timing side channels.

Recent work [30] has uncovered multiple vulnerabilities in real-world cryptographic libraries even under constrained speculative-execution models, such as only mispredicting limited types of branches. As modern processors evolve with ever-more-complex speculation mechanisms, we need mitigation solutions that protect broader speculative behaviors. Practical mitigation needs to navigate the complex trade-offs between security guarantees, performance overhead, and hardware complexity.

Many mitigation solutions [15, 18, 49, 56, 58, 63] share a common philosophy: identify secret data and then delay speculative execution for operations that may transmit such data. The key research challenge in these approaches lies in how to identify the secret data precisely without incurring high overhead. One promising design [18] is to augment the hardware with fine-grained taint tracking at the register level and coarse-grained taint tracking at the memory level (e.g., at page or section granularity). This architecture avoids the prohibitive costs of byte- or word-level tracking while retaining sufficient granularity to enforce secure speculation.

However, this hardware design requires the software to partition secret and public data into distinct memory regions explicitly, so that the hardware can interpret the secrecy status of the data accurately using its coarse-grained taint-tracking capability. Performance and security of the hardware design are contingent on precise annotation of secret data. Prior projects, ProSpeCT [18] and ConTExT [49], set out to add this partitioning capability to software through requiring fine-grained source-code annotations. Specifically, these methods require programmers to mark variables in the source code (e.g., C) as either secret or public. This information is then used as follows: For heap data, a customized memory allocator allocates secret and public objects in different memory pools. Stack data is protected by annotating secret and public stack variables manually to relocate them to different regions.

These source-level approaches work well for heap data but less so for the stack. Critically, they are unable to partition the stack accurately, requiring conservative partitioning and thereby suffering from performance loss. These limitations are *inherent* to source-level annotation methodologies. First, operating at the source level gives no visibility or control over register spills. Consequently, if a secret register is spilled to the stack, the programmer is forced to mark the whole stack as secret conservatively, leading to over-tainting and unnecessary performance overhead. Second, some approaches relocate secret data into global memory regions, which may compromise functional correctness under concurrency. Most

importantly, source-level transformations heavily rely on strong assumptions about compiler internals. However, given the complexity and opaqueness of modern compiler code bases, source-level transformation suffers from a significantly enlarged trusted computing base (TCB) and fragile compilation process that is difficult to verify.

1.1 This Paper

In this paper, we introduce *SecSep*, a transformation framework that rewrites assembly programs so that they partition secret and public data on the stack.

The key challenge in performing this assembly rewriting stems from loss of semantic information through the lengthy compilation process. The key innovation of our methodology is a new variant of typed assembly language (TAL), *Octal*, which allows us to address this challenge. *Octal* is specifically designed to enable *static* fine-grained taint tracking of assembly programs in software.

We design the type system by assigning dependent types and taint types to all registers and data objects in memory. The key idea is to leverage the dependent types to track the value ranges of registers and memory objects, so that we can construct the full picture of their points-to relationships throughout the program. While precise points-to analysis is infeasible for arbitrary programs, we take advantage of a domain-specific property, that is, cryptographic software is typically written following the constant-time programming discipline, making it amenable to our analysis. *Octal* also ensures well-typed programs are memory-safe.

Building upon *Octal*, we design a program-transformation framework *SecSep* consisting of two important components. The frontend is a heuristic type-inference algorithm that operates on off-the-shelf x86-64 assembly programs (plus debug tables already produced by Clang, plus a table mapping each function interface to a type in our new type system). The analysis involves a set of heuristic rules to reason about pointer arithmetic and loop counters. The outcome of the inference tool is an *Octal* program where the taint status of every memory operation is identified explicitly.

The backend of our framework is the code-transformation tool that rewrites programs based on taint annotations. It supports real-world cryptographic programs, which involve complex interleaving of secret and public reads/writes and shared pointer-based structures. After locating memory operations with taint types, their code is rewritten depending on their secrecy statuses.

We formally prove the type safety of *Octal*. We also prove that *SecSep*'s transformation separates secret and public data while guaranteeing functional correctness. We implement a hardware extension that achieves secure speculation with register-level and memory-segment-level taint tracking on the gem5 simulator [9, 38]. We evaluate *SecSep*'s transformation with the hardware extension using six cryptographic benchmarks [24] and show that it enables secure speculation with a negligible overhead of 1.2% on average.

In summary, we make the following contributions:

- We propose *Octal*, a variant of typed assembly language (TAL) with static fine-grained taint tracking for assembly programs.
- We design a program-transformation framework *SecSep* that (1) heuristically infers types for off-the-shelf cryptographic assembly programs and (2) rewrites them to split their secret and public data across coarse-grained memory regions.
- We prove soundness of the technique, provide a prototype implementation, and carry out an empirical evaluation showing low overhead.

Availability. Our prototype for *SecSep* is open-sourced at https://github.com/xx/xx.

2 BACKGROUND

2.1 Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks exploit transmitter instructions that leave visible side effects on microarchitectural state like caches [45, 59–61], TLBs [25], branch predictors [1, 21], and others [2, 5, 16, 20, 27, 28, 40, 47, 51, 55, 57]. Cryptographic programs prevent these attacks by following the constant-time coding discipline, which avoids executing such transmitter instructions with secret-dependent operands.

However, speculative-execution attacks [34–36, 39, 50] exploit the side effects of speculatively executed transmitters to leak the secrets, which are not blocked by the constant-time discipline.

2.2 Typed Assembly Language

Conventional assembly language omits most high-level semantic information, making static analysis challenging. Typed assembly language (TAL) [17, 23, 26, 41, 42] was introduced to regain some of that information, applied directly to assembly rather than source code. Such type systems have been demonstrated to guarantee properties like memory safety statically. In this paper, we adapt the concept of TAL for information-flow tracking to guide program transformation.

3 THREAT MODEL & SECURITY PROPERTIES

We aim to protect cryptographic applications against transient-execution attacks. Specifically, we assume the software is written following the constant-time programming discipline, in which the program when simulated using an abstract ISA machine avoids using secret-dependent values as branch conditions and memory addresses. However, the underlying hardware employs aggressive speculative-execution mechanisms, including prediction on both direct and indirect branches, which can result in transient instruction sequences that violate the constant-time assumption.

Our proposed assembly-rewriting technique is a key component in a software-hardware codesign mitigation. On the software side, our rewriting tool transforms the constant-time cryptographic programs to separate secret and public data into distinct regions. On the hardware side, we use an existing Spectre mitigation [18] with a fine-grained taint-tracking mechanism at the register level and coarse-grained taint tracking at the memory level, for a good trade-off between performance, cost, and security. The hardware uses the taint-tracking information to delay the execution of any potential transmitter instructions, which may leak information via timing side channels, when their operands are tainted.

We formalize the security property as a software-hardware contract as detailed below.

We use two observation models $[\![\cdot]\!]_{\text{pub}}$ and $[\![\cdot]\!]_{\text{ct}}$ to constrain the two software requirements listed above. Specifically, the notion

$[\![\cdot]\!]$ represents the observation trace of executing a program at the architectural level. Supposing the architectural trace of executing program P is $S_0 \xrightarrow{o_1} S_1 \xrightarrow{o_2} \ldots$, the observation trace is then defined as $[\![P]\!](S_0) = o_1 o_2 \ldots$.

Here, $[\![\cdot]\!]_{\text{pub}}$ records a trace of data values stored in the public memory region, and $[\![\cdot]\!]_{\text{ct}}$ records the trace of all load/store addresses and branch targets. We then define public noninterference, the software property guaranteed by *SecSep*, where $\simeq_{\text{pub}}$ constrains that the two states have equal values in the public memory region (see Definition 23).

Definition 1 (Software Public Noninterference). A program P satisfies *software public noninterference* for a specific public region if for all initial configurations S and S', if $S \simeq_{\text{pub}} S'$, then $[\![P]\!]_{\text{pub}}(S) = [\![P]\!]_{\text{pub}}(S')$ and $[\![P]\!]_{\text{ct}}(S) = [\![P]\!]_{\text{ct}}(S')$.

We use $\{\!|P|\!\}(S)$ to denote the microarchitectural observation trace of running program P on our out-of-order processor with initial state S [49]. The hardware must obey the following condition.

Definition 2 (Hardware Public Noninterference). A processor satisfies *hardware public noninterference* if for all program P and all initial states S, S', if $[\![P]\!]_{\text{pub}}(S) = [\![P]\!]_{\text{pub}}(S')$ and $[\![P]\!]_{\text{ct}}(S) = [\![P]\!]_{\text{ct}}(S')$, then $\{\!|P|\!\}(S) = \{\!|P|\!\}(S')$.

In summary, *SecSep* achieves secure speculation by ensuring that the software component satisfies the public noninterference contract. For the hardware component, we refer readers to the ProSpeCT paper [18], which provides formal proof that the taint-tracking hardware mechanism described above satisfies hardware public noninterference. Together, the software-hardware contract ensures end-to-end security of the overall system, where secret data do not influence microarchitectural side channels, even in the presence of speculative execution.

4 MOTIVATION AND OVERVIEW

In this section, we begin by discussing the limitations of prior work. We then provide an overview of our framework.

4.1 Limitations of Source-Level Annotation

A fundamental limitation of source-level code transformation lies in its heavy reliance on assumptions about compiler internals. Specifically, in ProSpeCT and ConTExT, programmers are required to annotate variables as secret manually at their declarations. When we mark a variable as "static," the compiler is expected to allocate the variable in the global region and mark the whole region as secret. However, this strategy fails to guarantee that the resulting code satisfies the public-noninterference property, due to the lack of control over register allocation, spilling, and compiler optimizations.

To illustrate how such a strategy can go wrong in practice, we present a case study from the `salsa20_words` function of a cryptographic library. Figure 1 shows both the annotated C code (Figure 1a) and the corresponding assembly code generated by clang-16 (Figure 1b). In the C code, the function takes a public variable `d` and a secret array `s[16]` as input. It then declares a local array `x[4][4]`, which is used to hold secret data from array `s` (lines 7-8) and is further used for computation in lines 11-13.

Given that array `x` holds secret data, a programmer can annotate it with the "static" keyword, expecting the compiler to allocate it in the secret-marked global region. However, the generated assembly code shown in Figure 1b deviates significantly from this expectation.

First, the compiler notices the array size is small enough to be stored in registers and decides to skip memory allocation for `x` completely. Specifically, in lines 6-7 of the assembly code, multiple elements inside the secret input array (base pointer `rsi`) are loaded into distinct registers, with no redirection to a secret region.

Second, we observe that in line 11, a secret register is spilled onto the stack, mixing secret data with many other public stack values. Since the source-level annotation has no control over register spilling, the programmer is forced to mark the whole stack as secret, resulting in overtainting and serious performance degradation. For example, according to our experiment on the `salsa20` application, this conservative approach results in 70% performance degradation.

4.2 Overview of *SecSep*

We propose *SecSep*, a framework to perform the secret-public memory separation at the assembly level. By operating after compilation, we have full control over memory layout. Our approach addresses the following two challenges.

First, we lack high-level semantic and pointer information. As high-level semantic information is lost during compilation, we need to recover information to identify which instructions operate on secret data and need to be transformed. A further complication is use of weakly typed pointers and potential pointer aliasing, which is particularly difficult to resolve without explicit type information. To deal with this challenge, we design a variant of typed assembly language called *Octal* and an inference algorithm to deduce type information for off-the-shelf x86-64 programs.

Second, we face the challenge of performing assembly transformation due to architectural constraints. Specifically, we are restricted from using extra registers, as this requires complex register management and register spilling. To deal with the challenge, we arrange our memory layout to have the secret region (i.e. secret stack) and the original stack (i.e. public stack) maintain a constant distance (δ) from each other. As a result, redirecting memory accesses between the stacks only requires pointer offsetting by δ.

To illustrate the effectiveness of our mechanism, we revisit the example in Figure 1. In Figure 1c, we show the type annotations derived by our inference tool. For brevity, we only show the memory-related annotations. Each memory operand is annotated with a *dependent type* that constrains its memory-access range and a *taint type* indicating secrecy. For example, in line 6, the array base pointer `rsi`, which references the secret input, is inferred to access the range of $[s, s + 64)$ with the taint type as 1, indicating secrecy. In line 11, another stack access is annotated with the access range as $[p - 88, p - 80)$ and is similarly marked as secret.

Transformation should relocate any pointer with taint type of 1. For example, in line 11, the stack offset is incremented by δ to move the write to the secret stack. Additionally, the parent function (not shown) adds δ to the base pointer `rsi` before passing it as an argument to the callee, ensuring all the accesses within the callee are redirected to the secret stack.

```c
void salsa20_words
(uint32_t *d, uint32_t s[16]) {
  __attribute__
  ((section("sec"))) static
  uint32_t x[4][4];
  int i;
  for (i=0; i<16; ++i)
    x[i/4][i%4] = s[i];
// Omit calculation on x
...
  for (i=0; i<16; ++i)
    d[i] = x[i/4][i%4] + s[i];
}
```

(a) Annotated source code

```
salsa20_words:

# Public spill
  movq %rdi, -64(%rsp)
# Source: line 7-8
  movl (%rsi), %r12d
  movl 4(%rsi), %r11d
  movl 8(%rsi), %r9d
  movl 12(%rsi), %eax
# Secret spill
  movq %rax, -88(%rsp)
...
```

(b) Original assembly code

```
salsa20_words:
# rdi: d, rsi: s, rsp: p
# Public spill
  movq %rdi, -64(%rsp)[p−64,p−56),0
# Source: line 7-8
  movl (%rsi)[s,s+64),1, %r12d
  movl 4(%rsi)[s,s+64),1, %r11d
  movl 8(%rsi)[s,s+64),1, %r9d
  movl 12(%rsi)[s,s+64),1, %eax
# Secret spill
  movq %rax, δ-88(%rsp)[p−88,p−80),1
...
```

(c) Transformed assembly code

Figure 1: Program transformation: source-code annotation v.s. assembly rewriting

The following sections now go into detail on the main components of our approach: type system (section 5), type inference (section 6), and transformation (section 7).

5 *OCTAL*

We propose *Octal*, a variant of typed assembly language [42] that helps reason about information flow statically. The abstract ISA machine for *Octal* applies taint tracking on registers and memory at the byte level. This machine tracks the secret flow and does not allow executing instructions that transmit tainted values through side channels. For example, it gets stuck when executing load/store with tainted addresses or branches with tainted conditions.

The goal of *Octal*'s type system is to ensure that a well-typed program and the program generated from it by our transformation are constant-time, thereby never getting stuck on this abstract machine. It is challenging to reason statically about the program's taint flow, since high-level abstractions such as pointers and array indices are missing in original x86-64 assembly programs. *Octal* enriches programs with types that not only constrain the taint status but also bound the values of registers and memory slots.

Furthermore, *Octal* splits memory into nonoverlapping slots according to the memory layout of the source program, associating a type to each memory slot. In this work, we only consider assembly programs compiled from constant-time C/C++ programs, so each memory slot contains either a scalar, pointer, or array, the last of which can have lengths not known at compile time, thanks to the use of symbolic descriptions of address ranges.

This design choice offers an additional benefit for information-flow tracking. Specifically, in cryptographic programs, although each static instruction may access different memory bytes during dynamic execution, it idiomatically only accesses data within the address range corresponding to a specific data object in the source program. Therefore, each static instruction in *Octal* programs has fixed registers/memory slots acting as its taint source and destinations, allowing easy regulation of taint flow statically with types.

In the remainder of this section, we first define *Octal* syntax in Section 5.1 and typing rules in Section 5.2. We also formalize type soundness in Section 5.3.

5.1 *Octal* Syntax

Program Syntax. *Octal* (selected syntax in Figure 2) is built based on x86-64. We require that each basic block end with **halt** to finish

$$
\begin{array}{rcll}
op & ::= & r \mid i \mid \ell \mid i_d(r_b, r_i, i_s)^{s,\tau} & \textit{Operand} \\
inst & ::= & \mathbf{movq}\ op_1, op_0 \mid \mathbf{leaq}\ op_1, op_0 & \textit{Instruction} \\
 & & \mid \mathbf{addq}\ op_1, op_0 \mid \mathbf{cmpq}\ op_1, op_0 & \\
 & & \mid \mathbf{jne}\ \ell^{\sigma} \mid \mathbf{jmp}\ \ell^{\sigma} \mid \mathbf{callq}\ \ell^{\sigma_{call},\sigma_{ret}} & \\
 & & \mid \mathbf{retq} \mid \mathbf{halt} & \\
I & ::= & \mathbf{jmp}\ \ell^{\sigma} \mid \mathbf{retq} \mid \mathbf{halt} & \textit{Instruction} \\
 & & \mid inst; I & \textit{sequence} \\
F & ::= & \{\ell_1 : I_1, \ldots, \ell_n : I_n, f_{\text{ret}} : \mathbf{retq}\} & \textit{Function} \\
P & ::= & \{f_1 : F_1, \ldots, f_n : F_n\} & \textit{Program} \\
 & & & \\
R & ::= & \{r_1 : (v_1, t_1), \ldots\} & \textit{Register file} \\
M & ::= & \{addr_1 : (v_1, t_1) \ldots\} & \textit{Memory} \\
S & ::= & (R, M, pc) & \textit{State} \\
 & & & \\
e & ::= & x \mid v \mid \top \mid e_1 \oplus e_2 \mid \ominus e & \textit{Dependent type} \\
\tau & ::= & x \mid 0 \mid 1 \mid \tau_1 \vee \tau_2 & \textit{Taint type} \\
\beta & ::= & (e, \tau) & \textit{Basic type} \\
\mathcal{R} & ::= & \{r_1 : \beta_1, \ldots\} & \textit{Register type} \\
\mathcal{M} & ::= & \{s_1 : (s_1^{\text{valid}}, \beta_1), \ldots\} & \textit{Memory type} \\
\mathcal{S} & ::= & (\Delta, \mathcal{R}, \mathcal{M}) & \textit{State type} \\
\Gamma & ::= & \{\ell_1 : (\Delta_1, \mathcal{R}_1, \mathcal{M}_1), \ldots\} & \textit{Function type} \\
\mathcal{P} & ::= & \{f_1 : \Gamma_1, \ldots\} & \textit{Program type}
\end{array}
$$

Figure 2: *Octal* syntax

execution or an unconditional branch. *Octal* also requires that each function f (except for the top-level one) has a basic block f_{ret} that only contains one return instruction to serve as the unique exit point for the function, which simplifies the typing rules.

Octal also introduces type annotations on load/store operands, branch instructions, and function calls (highlighted in blue in Figure 2). These annotations help to constrain well-formed programs, which will be detailed in Section 5.2.

Type Syntax. As mentioned before, an *Octal* abstract machine, with its machine state denoted as $S = (R, M, pc)$, applies byte-level taint tracking on the registers R and memory M and gets stuck on insecure operations (e.g., load/store with tainted addresses and branches with tainted conditions).

Octal's program type $\mathcal{P}$ is a map from function names to function types, and a function type Γ is a map from the function's basic-block labels to state types $\mathcal{S}$. A state type $\mathcal{S}$ serves as a precondition for

safe execution of a block.

A type $\mathcal{S}$ contains three parts: the type context Δ, register-file type $\mathcal{R}$, and memory type $\mathcal{M}$. Specifically, Δ is a set of constraints that must be satisfied by type variables in $\mathcal{R}$ and $\mathcal{M}$. Partial map $\mathcal{R}$ assigns register names to their dependent and taint types. A well-formed program can only read from registers that appear in $\mathcal{R}$. Partial map $\mathcal{M}$ assigns disjoint memory slots (s) each to a region of addresses whose contents are initialized (s^{valid}) and a type of data found therein. Each slot corresponds to a data object in the source program or a register spill. *Octal* tracks pointers in registers and memory using dependent types, and both memory slots s and valid regions s^{valid} are sets of addresses represented by dependent types. Hence, with the dependent types of load/store addresses, *Octal* can easily track which memory slot is accessed by each instruction.

5.2 Typing Rules

In *Octal*, program type-correctness is determined by the type-correctness of each function in the program, in turn determined by the type-correctness of each block in the function. Intuitively, the state type of each basic block ensures that the abstract machine whose state satisfies the type constraints can execute the block without getting stuck. When the machine is about to jump to the other block at a branch instruction, its state should also satisfy the target's state type. Figures 3 and 4 elaborate with typing rules.

In general, each of these instruction-sequence typing rules is structured as follows. First, the instruction sequence's state type should provide enough constraints so that the abstract machine can execute the first instruction in the sequence safely without getting stuck. Second, the rule derives new type constraints for the machine state after executing the first instruction. It requires that the next instruction sequence to be executed is well-typed with respect to the derived state type. Some examples illustrate the pattern.

Typing-Movq-m-r constrains a load instruction to be memory-safe and constant-time, via a type annotation tracking taint status of the load data. It invokes Typing-Load in Figure 4, which requires that the load range fall in the valid region in s ($s_{\text{addr}} \subseteq s^{\text{valid}}$), and the taint type of data in the slot must satisfy τ. *Octal* also requires the load address to be untainted, to guarantee constant time.

Typing-Movq-r-m handles store instructions. Similar to the rule for load, it checks the store address and store data's taint type against the store operand's annotation, derives the type of the corresponding memory slot after the store, and constrains the updated state type for the remaining instruction sequence. One thing to note is that *Octal* introduces two different strategies to constrain store-data taint and update the store-memory type.

When storing to a spill slot, as shown in Typing-StoreOp-Spill, *Octal* always derives the type for the next state by overwriting the slot's valid region and type with the store range and store data type. Even for a partial store, the type system "forgets" the type for the data in the part of the slot that is not overwritten by the operand.

When storing to a nonspill slot, as shown in Typing-StoreOp-Non-Spill, *Octal* requires the store data's taint status to satisfy the original taint type of the target memory slot. It also updates the valid region and dependent type by combining the store data and the existing data in the slot. For a partial store, *Octal* may only consider the updated dependent type as $\top$ for simplicity. Each nonspill slot corresponds to a data object in the source file, and so we impose uniformity on the taint type of the slot during its lifetime, i.e., the whole function. Then, we can use its taint type as a hint for its target placement during transformation, to change all load/store operands accessing it accordingly. We do not require the unified taint for register spills since we consider the register spill lifetime ends after the next register spill (or store) to the same slot.

Typing-Movq-m-r

$$\frac{\Delta, \mathcal{R}, \mathcal{M} \vdash \text{load}(i_d(r_b, r_i, i_s)^{s,\tau}, 8) : \beta \quad \mathcal{R}' = \mathcal{R}[r_0 \mapsto \beta] \quad \mathcal{P}, \Gamma \vdash I : (\Delta, \mathcal{R}', \mathcal{M})}{\mathcal{P}, \Gamma \vdash \textbf{movq}\, i_d(r_b, r_i, i_s)^{s,\tau}, r_0; I : (\Delta, \mathcal{R}, \mathcal{M})}$$

Typing-Movq-r-m

$$\frac{\Delta, \mathcal{R}, \mathcal{M} \vdash \text{store}(i_d(r_b, r_i, i_s)^{s,\tau}, 8, \mathcal{R}[r_1]) : (s^{\text{valid}}, \beta) \quad \mathcal{M}' = \mathcal{M}[s \mapsto (s^{\text{valid}}, \beta)] \quad \mathcal{P}, \Gamma \vdash I : (\Delta, \mathcal{R}, \mathcal{M}')}{\mathcal{P}, \Gamma \vdash \textbf{movq}\, r_1, i_d(r_b, r_i, i_s)^{s,\tau}; I : (\Delta, \mathcal{R}, \mathcal{M})}$$

Typing-Cmpq-r-r

$$\frac{\mathcal{R}[r_1] = (e_1, \tau_1) \quad \mathcal{R}[r_0] = (e_0, \tau_0) \quad \mathcal{P}, \Gamma \vdash I : (\Delta, \text{setFlag}(\mathcal{R}, (e_0 - e_1, \tau_0 \vee \tau_1), \textbf{cmpq}), \mathcal{M})}{\mathcal{P}, \Gamma \vdash \textbf{cmpq}\, r_1, r_0; I : (\Delta, \mathcal{R}, \mathcal{M})}$$

Typing-Jne

$$\frac{\begin{array}{c}\mathcal{R}[\texttt{ZF}] = (e = 0, 0) \\ \Delta \vdash \text{isNonChangeExp}(e) \quad \text{getInputVar}(\text{dom}(\sigma)) = \emptyset \\ \text{getTaintVar}(\text{dom}(\sigma)) = \emptyset \quad \forall x \in \text{dom}(\sigma).\ \sigma(x) \neq \top \\ \mathcal{P}, \Gamma \vdash I : (\Delta \cup \{e = 0\}, \mathcal{R}, \mathcal{M}) \quad \Gamma(\ell) = (\Delta', \mathcal{R}', \mathcal{M}') \\ \text{dom}(\mathcal{M}') = \text{dom}(\mathcal{M}) \quad (\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}) \sqsubseteq \sigma(\Delta', \mathcal{R}', \mathcal{M}')\end{array}}{\mathcal{P}, \Gamma \vdash \textbf{jne}\, \ell^{\sigma}; I : (\Delta, \mathcal{R}, \mathcal{M})}$$

Typing-Callq

$$\frac{\begin{array}{c}e = sp + c \quad \mathcal{R}[r_{\texttt{rsp}}] = (e, 0) \quad \mathcal{M}[e - 8, e] = (\emptyset, (\_, 0)) \\ \forall x, \sigma_{\text{call}}(x) = e, \text{isPtr}(x).\ \Delta \vdash \text{isNonChangeExp}(e - \text{getPtr}(e)) \\ \forall x \in \text{dom}(\sigma_{\text{call}}).\ \sigma_{\text{call}}(x) \neq \top \quad \sigma_{\text{ret}} = \overrightarrow{x_1} \to \overrightarrow{x_2} \\ \mathcal{R}_{p_0} = \mathcal{R}[r_{\texttt{rsp}} \mapsto (e - 8, 0)] \quad (\Delta, \mathcal{R}_{p_0}, \mathcal{M}) \sqsubseteq \sigma_{\text{call}}(\mathcal{P}(f)(f)) \\ \overrightarrow{x_2} \notin (\Delta, \mathcal{R}, \mathcal{M}) \quad (\Delta_{p_1}, \mathcal{R}_{p_1}, \mathcal{M}_{p_1}) = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(\mathcal{P}(f)(f_{\text{ret}})) \\ \mathcal{P}, \Gamma \vdash I : (\Delta \cup \Delta_{p_1}, \mathcal{R}_{p_1}[r_{\texttt{rsp}} \mapsto (e, 0)], \text{updateMem}(\mathcal{M}, \mathcal{M}_{p_1}))\end{array}}{\mathcal{P}, \Gamma \vdash \textbf{callq}\, f^{\sigma_{\text{call}}, \sigma_{\text{ret}}}; I : (\Delta, \mathcal{R}, \mathcal{M})}$$

Figure 3: Instruction-sequence typing

Typing-Jne specifies the rule for a conditional-branch instruction.The flag type is set by instructions such as **cmpq** (Typing-Cmpq-r-r). *Octal* requires the flag holding the branch condition to be untainted so that the program is constant-time. Furthermore, *Octal* also tracks whether each dependent type refers to pointer values which might be changed by our transformation (see Section 7.1 for details). To guarantee functional correctness of the transformation, *Octal* requires that the branch condition is independent from these pointer values, denoted as isNonChangeExp(e). Then, *Octal* derives the next state types after executing the branches, including both cases where the branch is taken and not taken.

For the not-taken side, similar to previous cases for non-branch instructions, *Octal* derives the next state type by adding the negation of the branch condition (i.e., $e = 0$) to the type constraints.

$$\textsc{Typing-Addr}\quad \frac{\mathcal{R}[r_b] = (e_b, \tau_b) \qquad \mathcal{R}[r_i] = (e_i, \tau_i)}{\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_b + e_i \times i_s + i_d, \tau_b \vee \tau_i)}$$

$$\textsc{Typing-Load}\quad \frac{\begin{array}{c}\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_{\text{addr}}, 0) \\ \Delta \vdash \text{isNonChangeExp}(e_{\text{addr}} - \text{getPtr}(s)) \\ s_{\text{addr}} = [e_{\text{addr}}, e_{\text{addr}} + c) \qquad \mathcal{M}[s] = (s^{\text{valid}}, (e, \tau)) \\ \Delta \vdash s_{\text{addr}} \subseteq s^{\text{valid}} \qquad e' = (\Delta \vdash s_{\text{addr}} = s^{\text{valid}})\ ?\ e : \top \\ e' = \top \Rightarrow (\Delta \vdash \text{isNonChangeExp}(e))\end{array}}{\Delta, \mathcal{R}, \mathcal{M} \vdash \text{load}(i_d(r_b, r_i, i_s)^{s,\tau}, c) : (e', \tau)}$$

$$\textsc{Typing-StoreOp-Spill}\quad \frac{\begin{array}{c}\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_{\text{addr}}, 0) \\ \Delta \vdash \text{isNonChangeExp}(e_{\text{addr}} - \text{getPtr}(s)) \\ s_{\text{addr}} = [e_{\text{addr}}, e_{\text{addr}} + c) \qquad s \in \text{dom}(\mathcal{M}) \\ \text{isSpill}(s) \qquad \Delta \vdash s_{\text{addr}} \subseteq s \qquad \Delta \vdash \tau_1 \Rightarrow \tau\end{array}}{\Delta, \mathcal{R}, \mathcal{M} \vdash \text{store}(i_d(r_b, r_i, i_s)^{s,\tau}, c, (e, \tau_1)) : (s_{\text{addr}}, (e, \tau))}$$

$$\textsc{Typing-StoreOp-Non-Spill}\quad \frac{\begin{array}{c}\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_{addr}, 0) \qquad s_{\text{addr}} = [e_{\text{addr}}, e_{\text{addr}} + c) \\ \Delta \vdash \text{isNonChangeExp}(e_{\text{addr}} - \text{getPtr}(s)) \\ \mathcal{M}[s] = (s^{\text{valid}}, (e_0, \tau)) \qquad \neg\text{isSpill}(s) \qquad \Delta \vdash s_{\text{addr}} \subseteq s \\ \Delta \vdash \tau_1 \Rightarrow \tau \qquad e' = (\Delta \vdash s^{\text{valid}} \subseteq s_{\text{addr}})\ ?\ e_1 : \top \\ e' = \top \Rightarrow (\Delta \vdash \text{isNonChangeExp}(e_0) \wedge \text{isNonChangeExp}(e_1))\end{array}}{\Delta, \mathcal{R}, \mathcal{M} \vdash \text{store}(i_d(r_b, r_i, i_s)^{s,\tau}, c, (e_1, \tau_1)) : (s_{\text{addr}} \cup s^{\text{valid}}, (e', \tau))}$$

Figure 4: Memory-operation typing (note that, via predicate isSpill, we rely on debug tables already generated by Clang)

For the taken side, *Octal* derives the next state type by asserting the branch condition (i.e., $e \neq 0$). The primary goal is to ensure that the machine state at the branch instruction is well-formed to jump to the target block. We define the subtype judgment for state types as shown in Figure 5. Intuitively, this judgment ensures that for any machine state S that satisfies a state type $\mathcal{S}_1$, if $\mathcal{S}_1$ is a subtype of $\mathcal{S}_2$, then S must also satisfy $\mathcal{S}_2$. Typing-Jne requires that the state type for the branch's taken side is a subtype of the target block's type $\Gamma(\ell)$. Note that in this rule, we are checking the subtype relation against $\sigma(\Gamma(\ell))$, where the branch annotation σ is a substitution that instantiates type variables in $\Gamma(\ell)$ using expressions over variables in the current block's type context. We use $\sigma(\cdot)$ as syntax sugar for applying the substitution σ to a variety of syntactic objects. Our type checker implements every entailment check $\Delta \vdash \ldots$ as a call to an SMT solver. In this rule, *Octal* also has some extra constraints on σ to guarantee type safety and transformation correctness, which will be detailed in Appendix A.1.

Typing-Callq specifies type constraints and changes of each step of calling a function. It first derives the state type $(\Delta, \mathcal{R}_{p_0}, \mathcal{M})$ after pushing the return address, checking that the state type is a subtype of the callee function's first block type $\mathcal{P}(f)(f)$ with respect to the function call's annotation σ_{call}. Here, σ_{call} represents the type-variable substitution between the callee and the caller.

$$\textsc{Reg-Subtype}\quad \frac{\Delta \vdash e_1 = e_2 \vee (\text{isNonChangeExp}(e_1) \wedge e_2 = \top) \qquad \Delta \vdash \tau_1 \Rightarrow \tau_2}{\Delta \vdash (e_1, \tau_1) \sqsubseteq (e_2, \tau_2)}$$

$$\textsc{Mem-Slot-Subtype}\quad \frac{\begin{array}{c}\Delta \vdash s_2 \subseteq s_1 \qquad \Delta \vdash \text{isSpill}(s_2) \Rightarrow \text{isSpill}(s_1) \\ \Delta \vdash e_1 = e_2 \vee (\text{isNonChangeExp}(e_1) \wedge e_2 = \top) \vee s_2 = \emptyset \\ \Delta \vdash \tau_1 = \tau_2 \vee (\text{isSpill}(s_1) \wedge s_2 = \emptyset) \qquad \text{getPtr}(s_1) = \text{getPtr}(s_2)\end{array}}{\Delta \vdash (s_1, (e_1, \tau_1)) \sqsubseteq (s_2, (e_2, \tau_2))}$$

$$\textsc{State-Subtype}\quad \frac{\begin{array}{c}\Delta_1 \vdash \Delta_2 \qquad \forall r \in \text{dom}(\mathcal{R}_2).\ \Delta_1 \vdash \mathcal{R}_1[r] \sqsubseteq \mathcal{R}_2[r] \\ \forall s_2 \in \text{dom}(\mathcal{M}_2).\ \exists s_1.\ \Delta_1 \vdash (s_2 \subseteq s_1 \wedge \mathcal{M}_1[s_1] \sqsubseteq \mathcal{M}_2[s_2])\end{array}}{\vdash (\Delta_1, \mathcal{R}_1, \mathcal{M}_1) \sqsubseteq (\Delta_2, \mathcal{R}_2, \mathcal{M}_2)}$$

Figure 5: State subtyping

Next, *Octal* derives the state type after returning from the callee, using the type of the callee's exit block. There are several details to note. First, we need to convert the return-state type represented under the callee's type context to the caller's context. Compared to the callee's first block type $\mathcal{P}(f)(f)$, its return-state type $\mathcal{P}(f)(f_{\text{ret}})$ may introduce new type variables. The type annotation σ_{ret} maps these new variables to the caller's context. Hence, we perform type-variable substitution using both substitutions to represent the return-state type under the caller's context, i.e., $(\Delta_{p_1}, \mathcal{R}_{p_1}, \mathcal{M}_{p_1}) = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(\mathcal{P}(f)(f_{\text{ret}}))$. We then add the return state's type constraints Δ_{p_1} to the next state type's context. Second, the callee's return-state type only specifies how it updates the memory region covered by its memory type, which is a subset of the memory region covered by the caller's memory type. On the other hand, according to our typing rules for load and store operations, the memory regions that do not belong to the callee's memory type remain unchanged across the function call. Following this philosophy, we apply the callee's changes to memory slots to the parent's memory type to get the final memory type after return $(\text{updateMem}(\mathcal{M}, \mathcal{M}_{p_1}))$. We also pop the return address to get the final return-state type.

5.3 Type Soundness

In this section, we justify the type safety of *Octal* programs, that is, *Octal* guarantees well-typed programs to be executed on an *Octal* abstract machine without getting stuck. First, we briefly explain well-formedness of *Octal* abstract machine states (see details in Appendix A.2). A state S is well-formed, i.e. $P, \mathcal{P} \vdash_{\text{TAL}} S$, if all its registers and memory values satisfy constraints specified by the state type of the instruction sequence to be executed next.

Theorem 3 (Type Safety). *If $P, \mathcal{P} \vdash_{TAL} S$, then for some S', $S \rightarrow S'$ and $P, \mathcal{P} \vdash_{TAL} S'$; or S is a termination state.*

6 TYPE INFERENCE

In this section, we introduce our type-inference algorithm that generates types for assembly programs. Note that the inference algorithm is heuristic and does not guarantee type correctness.

```
/**
 * @anno message : @size(mlen), @valid(0, mlen);
 * @anno mlen    : @taint[0];
 */
void foo(uint8_t *message, uint64_t mlen) { ... }
```

Figure 6: Example type annotations in C source code, which means that (1) the pointer `message` points to an array with size `mlen`, and the whole array is initialized; (2) `mlen` is a public variable whose taint type is 0.

Instead, the correctness is checked separately by applying typing rules introduced in Section 5. Recalling the definitions from Section 5.1, we need to generate state types at basic blocks and type annotations on instructions. Our type-inference algorithm consists of three parts. First, we introduce unification type variables to represent state types and type annotations. Second, we plug the type expressions into *Octal*'s typing rules to collect type constraints. Then, the third step is to solve for arithmetic predicates on type variables, which will be used to enrich the Δ of each block's state type so that it satisfies the typing constraints. We iterate a process of learning new type information and exploring its implications.

6.1 Type initialization

We start our type-inference process by using type-unification variables to represent state types of each basic block (i.e., $(\Delta, \mathcal{R}, \mathcal{M})$) and type annotations (i.e., load/store's destination-slot taint annotations; type-variable substitution annotations for branches and calls). Our goal is to add appropriate constraints on these type variables to the type context Δ so that the state types and annotations satisfy the typing rules in Section 5.2.

For register types, we simply assign a unification variable to each register; and for type annotations, we follow a similar strategy. For memory type $\mathcal{M}$, recall that it is a map of memory slots available to the function, and each slot corresponds to a data object in the source code or a register spill. To initialize $\mathcal{M}$, we first need to figure out $\mathrm{dom}(\mathcal{M})$ for each function.

Determine Memory Layout. Core cryptographic routines usually do not allocate memory on the heap dynamically for reasons of both performance and side-channel security, so we consider the following three kinds of memory slots to determine each function's memory layout: (1) data objects referenced by pointers in the function arguments; (2) local stack referenced by the stack pointer; (3) global variables referenced by global pointers.

We require simple type annotations in C source code, implemented through our custom annotation system, to explain the relationships among function parameters, e.g., one gives the size of an array that another points to (see example in Figure 6). These annotations are compiled down to assembly and serve as provided specifications for functions. However, we must infer type information for other basic blocks within each function. Other information can be determined with simple compiler support: address ranges of function stack frames using a Clang pass, and locations of global variables indicated directly in assembly code.

Constraint-Movq-m-r-Unknown

$$\frac{\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_a, \tau_a) \qquad \mathcal{R}' = \mathcal{R}[r_0 \mapsto (\top, \tau)] \qquad \mathcal{P}, \Gamma \vdash I : (\mathcal{R}', \mathcal{M}, \Delta) \Rrightarrow C}{\mathcal{P}, \Gamma \vdash \mathbf{movq}\ i_d(r_b, r_i, i_s)^{s,\tau}, r_0; I : (\Delta, \mathcal{R}, \mathcal{M}) \Rrightarrow [e_a, e_a + 8] \subseteq s; s \in \mathrm{dom}(\mathcal{M}); \tau_a = 0; C}$$

Constraint-Movq-r-m-Unknown

$$\frac{\mathcal{R} \vdash i_d(r_b, r_i, i_s) : (e_a, \tau_a) \qquad \mathcal{P}, \Gamma \vdash I : (\Delta, \mathcal{R}, \mathcal{M}) \Rrightarrow C}{\mathcal{P}, \Gamma \vdash \mathbf{movq}\ r_1, i_d(r_b, r_i, i_s)^{s,\tau}; I : (\Delta, \mathcal{R}, \mathcal{M}) \Rrightarrow [e_a, e_a + 8] \subseteq s; s \in \mathrm{dom}(\mathcal{M}); \tau_a = 0; C}$$

Constraint-Jne

$$\frac{\mathcal{R}[\mathtt{ZF}] = (e = 0, \tau) \qquad \mathcal{P}, \Gamma \vdash I : (\Delta \cup \{e = 0\}, \mathcal{R}, \mathcal{M}) \Rrightarrow C}{\mathcal{P}, \Gamma \vdash \mathbf{jne}\ \ell^{\sigma}; I : (\Delta, \mathcal{R}, \mathcal{M}) \Rrightarrow \tau = 0; (\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}) \sqsubseteq \sigma(\Gamma(l)); C}$$

Figure 7: Typing-constraints generation

6.2 Type-Constraint Generation

In this section, we describe rules to generate constraints on the initialized block-state types. We provide several example rules in Figure 7, where the generated constraints are highlighted. Given a state type and the corresponding instruction sequence, the constraint-generation rule consists of two parts, following a similar structure to *Octal* typing rules.

First, a rule generates constraints on the state type so that the current instruction executes safely. For example, the first two rules in Figure 7 constrain that a load/store operation must access a memory slot from the memory type, and the address must be untainted. The third rule requires that the branch condition is untainted.

Second, a rule derives the next state type after executing the first instruction in the sequence and generates constraints for the next type. Note that the state types are initialized using unification variables not constrained by predicates, so we may not be able to derive the next state type deterministically. For example, as shown in Constraint-Movq-r-m-Unknown, we cannot determine the target slot of the store operand and thereby are not able to update the memory type correspondingly. In this case, we use the unmodified memory type to generate constraints for the next instruction sequence. We acknowledge that these heuristic rules cannot generate all proper type constraints. We rely on these partially correct constraints to derive predicates and use the newly solved predicates to improve constraint generation in the next round.

6.3 Dependent Type Inference

In this section, we show how we derive arithmetic predicates of dependent type variables from type constraints. As shown in Figure 7, we generate two kinds of constraints for dependent types: (1) state-subtype and (2) load/store-address constraints.

6.3.1 Solving Subtype Constraints. *Octal* requires that the state type at each branch should be a subtype of the target block's state type, (e.g., $(\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}) \sqsubseteq \sigma(\Gamma(\ell))$ in Constraint-Jne). Intuitively, the subtype relation requires that the range of each

register/memory slot's value at the target block, represented by dependent type variables, should be a superset of the range of its value at the branch that jumps to the target block. By unfolding all subtype constraints, we can get concrete constraints on the range of each dependent type variable. We propose a set of inference rules that syntactically applies to the range constraints with certain patterns and solves the predicates of each variable heuristically.

Our heuristic rules for solving subtype constraints are built upon several common patterns we observed from the range constraints on each type variable. Recall we introduce dependent types in *Octal* to reason about load/store operations, so we only focus on type variables used for pointer arithmetic. Luckily, we target type inference for cryptographic programs, and their dependent-type range constraints share simple and intuitive patterns corresponding to two basic code patterns in Figure 8. In both examples, we demonstrate applying our rules to figure out the range for type variable a that represents `rax`'s dependent type at basic block `.L0`. We denote the range of a as S_a and derive constraints on S_a by unfolding all state subtype constraints.

Infer set of values. The first example (Figure 8a) shows the case where `rax` contains different values when entering basic block `.L0` from different branches. Specifically, the range of `rax` is $\{e_1\}$ when jumping from `.L1` and is $\{e_2\}$ when jumping from `.L2`. The subset constraint and the derived solution can be formulated as follows:

$$\left.\begin{array}{l} S_a \supseteq \{e_1\} \\ S_a \supseteq \{e_2\} \end{array}\right\} \Rightarrow S_a = \{e_1, e_2\}.$$

Infer range of loop counter. The second example (Figure 8b) shows the case where `rax` acts as a loop counter. `rax` is initialized to e_0 when entering the loop-body block `.L1` and increased by a constant step c_0 in each iteration. The loop ends when `rax` is equal to the boundary value e_n. Without loss of generality, we discuss the case where $c_0 > 0$. According to our constraint-generation rules, when jumping back to the loop head `.L0`, the state type satisfies $\Delta = \{a \in S_a, a + c_0 - e_n \neq 0\}$ and $\mathcal{R} = \{\texttt{rax} : a + c_0\}$. So we can use the set $\{a + c_0 : a \in S_a \wedge a + c_0 - e_n \neq 0\}$ to represent the range of `rax` before jumping back. Thus, the subtype constraint and the corresponding heuristic rule can be formulated as follows:

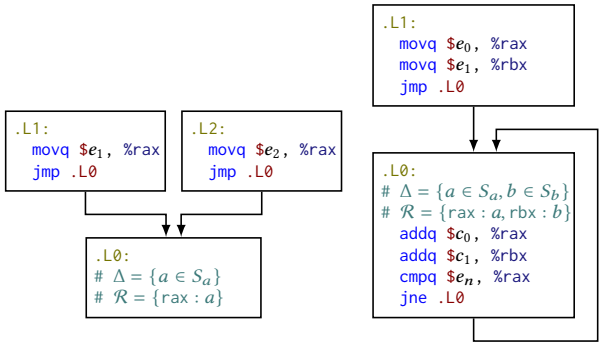
$$\left.\begin{array}{ll} S_a \supseteq \{e_0\} & c_0 > 0 \\ S_a \supseteq \{a + c_0 : a \in S_a \wedge a + c_0 \neq e_n\} & \end{array}\right\} \Rightarrow S_a = [e_0, e_n - c_0]_{c_0}.$$

This rule extracts three key features from the constraints:

- e_0: the loop counter's base value at the loop's entrance;
- c_0: the per-iteration step value for the counter;
- e_n: the loop boundary in the branch condition.

Then, the rule heuristically determines that the range of a is $S_a = [e_0, e_n - c_0]_{c_0}$. Here, we use $[a, b]_c$ to represent a set of values in range $[a, b]$ with stride c. In our implementation, we apply the above strategy to infer a range of loop counters.

Infer implicit relation between variables. Another challenge is that assembly programs do not explicitly keep semantic relations between type variables. However, these relations are crucial to deriving accurate range constraints for variables. For example, in Figure 8b, `rax` and `rbx` are increased consistently during each loop iteration, but the loop condition only constrains the boundary of `rax` when jumping back to the loop header. By unfolding the subtype constraints, we can only get the following constraints related



(a) Infer set of values **(b) Infer loop counter**

Figure 8: Examples for dependent type inference

```
1  foo:
2  # R = {rdi : p, rsi : n}
3  # M = {[p, p + n] : _}
4    cmpq $8, %rsi
5    jae .L0
6    retq
7  .L0:
8  # Missing predicate:
9  # Δ = {n ≥ 8}
10   movq (%rdi), %rax
```

(a) Boundary check

```
foo:
# foo (uint64_t p[8],
#      uint64_t k);
# R = {rdi : p, rsi : k}
# M = {[p, p + 64] : _}
# Missing predicate:
# Δ = {k ∈ [0, 7]}
  movq (%rdi, %rsi, 8), %rax
```

(b) Implicit assumption

Figure 9: Missing predicates to validate memory accesses

to b: $S_b \supseteq \{e_1\}$ and $S_b \supseteq \{b + c_1 : b \in S_b\}$, which implies that S_b is infinite, thereby not accurately constraining the range of b.

To solve this problem, we introduce another rule that infers the linear relation between type variables that share similar range-constraint patterns. In our example, `rax` and `rbx` are increased synchronously following the same loop structure, so we can use the range of `rax` to constrain the range of `rbx`, as shown in the following formula. See the example application below.

$$\left.\begin{array}{l} S_b \supseteq \{e_1\} \\ S_b \supseteq \{b + c_1 : b \in S_b\} \\ S_a \supseteq \{e_0\} \\ S_a \supseteq \{a + c_0 : a \in S_a \wedge a + c_0 \neq e_n\} \end{array}\right\} \Rightarrow S_b = \left\{\frac{(a - e_0)c_1}{c_0} + e_1 : a \in S_a\right\}.$$

6.3.2 Solving load/store constraints. *Octal* also requires that the dependent type of each load/store address belong to a specific memory slot. These constraints can be satisfied automatically with the predicates derived from the subtype relation when the load/store address and the memory slot have simple formulas, e.g., shifted from the base pointer by a constant offset. However, when accessing an array with a variable length or a variable index, we need extra predicates for bounds checks regarding the length/index type variables, inferred by the following two methods.

Propagate branch conditions. First, the function may already include proper bounds checks to guarantee memory safety. For example, as shown in Figure 9a, line 10 loads from $[p, p + 8]$, and we lack the predicate $n \geq 8$ to validate it. On the other hand, the program checks the branch condition $n \geq 8$ before jumping to `.L0`, which implies this missing predicate. Motivated by this common

pattern, we propose the following rule to deduce predicates by propagating branch conditions across basic blocks.

$$\left.\begin{array}{l}\sigma_1(\Delta,\mathcal{R},\mathcal{M}) \sqsupseteq (\{\sigma_1(e)\}\cup\Delta_1,\mathcal{R}_1,\mathcal{M}_1)\\ \sigma_2(\Delta,\mathcal{R},\mathcal{M}) \sqsupseteq (\{\sigma_2(e)\}\cup\Delta_2,\mathcal{R}_2,\mathcal{M}_2)\\ \dots\end{array}\right\} \Rightarrow e\in\Delta$$

Each subtype constraint listed here corresponds to one branch that jumps to the specific block with state type $(\Delta,\mathcal{R},\mathcal{M})$. This rule states that for all branches that jump to this block, if a predicate is always satisfied before branching, then it can be added to the block's type context.

Reverse-engineer load/store operations. However, not all missing predicates can be deduced from branch conditions in the function. For example, as shown in Figure 9b, the function takes two inputs: pointer p to an array with 8 entries and index k. The programmer implicitly assumes that the function is only called with $k\in[0,7]$ and loads from the kth entry of p without performing any boundary checks. We propose a two-step method to infer these implicit assumptions by reverse-engineering the necessary conditions to validate the memory safety of load/store operations.

First, for each load/store address without any known target memory slot, we heuristically guess which slot it belongs to based on its address pattern. For example, it is expected to belong to a memory slot that shares the same base pointer.

Second, we constrain the load/store operation to fall in the slot by adding the corresponding predicates to the current block's state type. As required by subtype constraints, this newly generated predicate must also be satisfied by every previous basic block that jumps to the current one. Hence, we apply the following rule to propagate each newly generated predicate to the previous blocks.

$$\left.\begin{array}{l}\sigma_1(\{e\}\cup\Delta,\mathcal{R},\mathcal{M}) \sqsupseteq (\Delta_1,\mathcal{R}_1,\mathcal{M}_1)\\ \sigma_2(\{e\}\cup\Delta,\mathcal{R},\mathcal{M}) \sqsupseteq (\Delta_2,\mathcal{R}_2,\mathcal{M}_2)\\ \dots\end{array}\right\} \Rightarrow \begin{cases}\sigma_1(e)\in\Delta_1\\ \sigma_2(e)\in\Delta_2\\ \dots\end{cases}$$

Note that both inference strategies require us to substitute local type variables properly for each basic block, i.e., to know the branch annotation σ. This type-variable substitution can be built by unifying each register and memory slot's type from the target block's state type and the state type before branching.

6.4 Valid-Region Inference

In this section, we explain how to infer valid regions of each basic block's state type, which is constrained by two things: (1) each load instruction can only read from valid regions (Typing-Load); (2) each memory slot's valid region at a branch instruction must be a superset of its valid region at the destination block (Mem-Slot-Subtype). Our overall inference strategy is to constrain the valid region of each memory slot using (2) and find the most accurate solution that covers the largest valid region to satisfy (1). Specifically, the second constraint can be derived from subtype constraints. For example, for $\sigma_1(\Delta,\mathcal{R},\mathcal{M}) \sqsupseteq (\Delta_1,\mathcal{R}_1,\mathcal{M}_1)$ and memory slot s where $\mathcal{M}[s] = (s^{\text{valid}},\_)$, $\mathcal{M}_1[s] = (s_1^{\text{valid}},\_)$, the constraint on s^{valid} is $\sigma_1(s^{\text{valid}}) \subseteq s_1^{\text{valid}}$.

For a memory slot that is fully initialized at the beginning of the function, its valid region is always equal to its address range. It is also straightforward to infer the valid region for a memory slot that holds a primitive type of data (e.g., `int`) or a register spill since the program usually writes to the full slot or leaves the full slot uninitialized. Hence, its valid region is usually the slot address range or the empty set. The major challenge is to infer the valid region for an array, where the program writes to part of it at a time, steadily increasing its valid region. We provide heuristic rules to represent the valid region accurately using dependent type variables.

```
1  .L1: # R = {rdi : p},  M = {[p, p+64] : (∅, _)}
2    movq $0, %rax
3    jmp .L0
4  .L0: # R = {rdi : p, rax : a},  M = {[p, p+64] : (s^valid, _)}
5       # Δ = {a ∈ [0, 63]}
6    movb %rsi, (%rdi, %rax)
7    addq $1, %rax
8    cmpq $64, %rax
9    jne .L0^σ # σ(a) = a + 1
```

Figure 10: Infer the valid region of an array

For example, as shown in Figure 10, the program fills an array by looping over all its entries. We can derive the following constraints on the array's valid region at `.L0`.

$$\left.\begin{array}{ll}[0/a]s^{\text{valid}} \subseteq \emptyset & a\in[0,63]\\ [a+1/a]\,(s^{\text{valid}}) \subseteq s^{\text{valid}}\cup[p+a,p+a+1] & \end{array}\right\} \Rightarrow s^{\text{valid}} = [p,p+a]$$

Our inference algorithm extracts the valid region's boundary from the pattern $s^{\text{valid}}\cup[p+a,p+a+1]$. The key insight is that the next array write is always to the next uninitialized slot.

6.5 Taint-Type Inference

In this section, we demonstrate how to unify local taint variables at each block with each function's input taint variables and generate necessary predicates to satisfy all constraints.

According to Section 5.2 and Section 6.2, *Octal* constrains taint types via the following three aspects:

(1) Load/store addresses and branch conditions are untainted. Denote the taint type of the address or the condition as $\tau = x_1\vee x_2\vee\dots x_n$. Then, we can rewrite the constraint as $x_1\Rightarrow 0\wedge x_2\Rightarrow 0\wedge\dots x_n\Rightarrow 0$.
(2) The taint type of the accessed memory slot is equal to the taint annotation of a load/store operand (under some scenarios). According to type-constraint generation, both the memory slot's taint type and the load/store operand's taint annotation, denoted as x_{slot} and x_{op}, are only represented by taint variables or constant taint values (instead of complex taint expressions). Therefore, the taint constraint can be written as $x_{\text{slot}}\Rightarrow x_{\text{op}}\wedge x_{\text{op}}\Rightarrow x_{\text{slot}}$,
(3) If store data is tainted, then the store operand's taint annotation is also tainted. Denote the store data's taint type as $x_1\vee x_2\vee\dots\vee x_n$ and the store operand's taint type as x_{op}. We can write the constraint as $x_1\Rightarrow x_{\text{op}}\wedge x_2\Rightarrow x_{\text{op}}\wedge\dots\wedge x_n\Rightarrow x_{\text{op}}$.

In short, the taint constraints can be summarized in the form $E_1\wedge E_2\wedge\dots\wedge E_n$. Here, each E_k has the form $x_1\Rightarrow x_2$ where x_1 and x_2 are either taint variables or constant taint values. This formula clearly constrains the taint flow among all taint variables.

Then, for each local taint variable, we can identify its taint source represented by input taint variables. If there is no taint source, we set it to 0. Otherwise, we set it to the logical OR of all its taint sources. We can also collect predicates for input taint variables in

the similar form $x_1 \Rightarrow x_2$ and add them to the state type of the function's input block.

7 TRANSFORMATION

We define a transformation that takes a well-typed *Octal* program as input and generates another program that satisfies our software contract, public noninterference (defined in Section 3). As discussed in Section 4.2, the overall strategy of our transformation is to maintain a secret stack that is shifted from the original stack by δ bytes. If a stack slot contains secrets, we shift its location by δ to move it onto the secret stack. If a stack slot contains public data, we do not change its location. Note that our transformation does not affect heap/global variables, while we do use *Octal*'s type system to ensure that they are used properly from an information-flow perspective. We first present two basic transformation strategies to relocate memory data. Then, we illustrate how we apply these two strategies to transform the program. We also formally prove that the transformation maintains the original program's functionality while guaranteeing public noninterference.

7.1 Two Memory-Relocation Strategies

Load/store instructions in x86-64 (and other ISAs) support the following addressing mode: taking a precalculated base pointer and adding an offset to the pointer to derive the target address. There are thus two basic applicable strategies to transform memory accesses in assembly programs:

TransPtr We can modify the base pointer before it is used in the memory operand so that the memory operand automatically switch to accessing the relocated object.

TransOp When we want to shift the target address by a constant offset, we can directly modify the memory operand to add this offset.

However, each strategy has limited applicability. First, `TransOp` is a context-insensitive change, which uniformly shifts the memory-access address by the same offset. As a result, `TransOp` is only suitable for the case where we statically know how the data object accessed by the instruction should be relocated (e.g., whether to move it to the secret stack). However, the program may reuse the same instruction to operate on public and secret data in different situations. For example, the `memset` function might be called to set either public or secret data objects, where we want to relocate them with different offsets. Note that on the caller side, we may know more context information such as whether the data object is secret or not. Hence, we choose `TransPtr` rather than `TransOp` to transform those store instructions in `memset` by modifying the pointer argument passed to `memset`, so that all the store instructions can automatically access the designated region.

On the other hand, when applying `TransPtr` to shift a base pointer, all load/store operands using the same base pointer will shift their target addresses by the same offset. In other words, all memory slots referenced by the same base pointer will be relocated together by `TransPtr`, so it is only suitable for the case where those referenced slots share the same taint type. For example, a function may access a struct that contains both secret and public fields (slots) through the same base pointer of the struct. In this case, we use `TransOp` to avoid relocating the public slots.

One important case worth discussing is about translating memory accesses to slots referenced by the stack pointer. On the one hand, the stack pointer, stored in `rsp`, is used to reference different memory slots on the function's local stack, including both tainted and untainted ones. So, we should not apply `TransPtr` to transform the stack pointer. On the other hand, the program may pass base pointers of stack objects to functions such as `memset`. These pointers are stored in registers such as `rdi` according to x86-64's calling convention, and they are only used to access the corresponding data objects instead of arbitrary slots on the stack. Hence, although the pointer points to the stack, we can still apply `TransPtr` as long as all slots within the corresponding object share the same taint.

7.2 Transformation Details

Determine transformation strategy. The first step of our transformation is to decide which transformation strategy to use for each memory access. We first determine the strategy for each memory slot and transform all memory accesses to that slot with the slot's strategy. Given a function with input memory type $\mathcal{M}$, we generate a map $\omega : \mathrm{dom}(\mathcal{M}) \to \{\texttt{TransOp}, \texttt{TransPtr}\}$ that maps each memory slot to its transformation strategy.

Following the discussion of the pros and cons of `TransPtr` and `TransOp` in Section 7.1, we propose the the following approach to decide which strategy to use: $\omega(s) = \texttt{TransPtr}$ if and only if (1) s is referenced by a pointer passed through a function argument, and (2) all slots in $\mathcal{M}$ referenced by the same pointer have the same taint type.[1] We formalize the generation of ω in Appendix B.1.

Transform load/store operands. Next, our transformation uses a pass C_{op} to transform all load/store operands that access memory slots whose transformation strategy is `TransOp`. For each memory operand, denoted as $i_d(r_b, r_i, i_s)^{s,\tau}$, s is the memory slot accessed by the operand, and τ is the slot taint type. If $\omega(s) = \texttt{TransOp}$ and $\tau \neq 0$ (i.e., the slot might be tainted), C_{op} will rewrite the operand to $\delta + i_d(r_b, r_i, i_s)$ so that its target address is shifted by δ and relocated to the secret region.

For instructions that perform load/store without explicit load/store operands in their ISA representations (e.g., **pushq**, **popq**), C_{op} also synthesizes the transformed behavior accordingly. Specifically, for simplicity, we will use **pushsecq** and **popsecq** to represent push/pop on the secret stack, which will be synthesized to valid x86-64 instructions in the final transformed program.

Transform pointer arguments. We define another pass C_{ptr} to perform `TransPtr`, which transforms pointer arguments passed to each callee function accordingly so that they use the transformed pointer to access designated regions.

Specifically, for each base pointer argument that references tainted slots with transformation strategy `TransPtr`, the callee function expects that the pointer is already shifted by δ and points to the address after relocation.

From the caller function's perspective, if the transformation strategy of the slots referenced by the same pointer is also `TransPtr`, then the pointer must be already shifted, and we do not need to make any changes. On the other hand, if the transformation strategy of those slots is `TransOp`, then the pointer is not transformed yet,

[1] We also require for slot s where its base pointer is a function argument and $\omega(s) = \texttt{TransOp}$, its taint type is 0 or 1.

```
1  fchild:
2    pushsecq %r12
3    ...
4    popsecq %r12
5  # r12 is
6  # from tainted
7  # stack
8
```

(a) r12 → tainted

```
fparent_sec:
  pushsecq %r12^{s+δ,τ}
  movq $sec, %r12
  callq child
  popsecq %r12^{s+δ,τ}
```

(b) Secret r12

```
fparent_pub:
  pushsecq %r12^{s+δ,τ}
  movq $ptr, %r12
  movq %r12, (%rsp)^{s,0}
  callq child
  movq (%rsp)^{s,0}, %r12
  movq %rax, (%r12)
  popsecq %r12^{s+δ,τ}
```

(c) Public r12

Figure 11: Restore callee-saved registers' taint

so we need to add δ to the pointer argument when passing it to the callee. We formalize this transformation in Appendix B.1.

Restore callee-saved registers' taint. With C_{op} and C_{ptr}, our transformation can ensure that all memory operands accessing secret data on the stack are redirected to accessing the secret stack. However, recall that `TransOp` shifts a memory operand's target address to the secret stack as long as the corresponding memory slot is not constantly untainted. In other words, our transformation may conservatively redirect memory accesses to the secret stack even though they may operate on the public data under some circumstances, causing performance loss.

Specifically, each function usually saves callee-saved registers to its stack if needed, and restores them before returning to the call site. Since the callee-saved registers can be tainted or untainted depending on the call site, we transform the function to always push them to the secret stack to avoid potential leakage. For example, in Figure 11, `fparent_sec` calls `fchild` with one callee-saved register `r12` containing secret data, while `fparent_pub` calls `fchild` with `r12` containing a public pointer. We then transform `fchild` to save `r12` to the secret stack. As a result, when returning to `fparent_pub` after calling `fchild`, `r12` is marked as tainted by a processor with coarse-grained memory taint tracking and secure speculation. Since `r12` is used as the store address on line 7, and the processor delays speculative memory accesses with tainted addresses to avoid leaking secrets, this store will be delayed until the commit stage, hurting performance.

We introduce another pass C_{callee} that saves public callee-saved registers to the public stack before each function call and retrieves them after the call. C_{callee} guarantees that when running the transformed program on a machine that does coarse-grained taint tracking, the callee-saved registers are never overtainted after function calls.

Extra stack space need not be allocated. As shown in Figure 11c, `fparent_pub` pushes `r12` to the secret stack slot $s+\delta$ before using it, while the corresponding slot s on the public stack is unused. Thus, we can use s to save and restore the public value in `r12` before and after calling `fchild` (highlighted in Figure 11c). We formalize C_{callee} in Appendix B.2.

Note that although C_{callee} helps avoid unnecessary delays on speculation, it inserts extra instructions to the program and may cause performance overhead. We will evaluate this tradeoff by measuring the performance of our defense with and without C_{callee} in Section 8.2.

7.3 Transformation Soundness

Functional Correctness. First, we define a simulation relation $P, \mathcal{P} \vdash S' \prec S$, which correlates abstract machine states running the transformed program and the original program P. Intuitively, the simulation relation maps the relocated memory data objects in the transformed program's state to those in the original program's. It also requires that paired objects and registers in the two states have matching values as long as they are not pointers that might be changed by `TransPtr`.

Denoting our overall transformation as C, we can formalize the functional correctness of our transformation using the following theorem.

Theorem 4 (Functional Correctness). *If $P, \mathcal{P} \vdash_{TAL} S$ and $P, \mathcal{P} \vdash S' \prec S$, then there exists S_1 and S_1' such that $S \xrightarrow{inst} S_1$, $S' \xrightarrow{C(inst)}{}^{*} S_1'$, and $P, \mathcal{P} \vdash S_1' \prec S_1$; or S and S' are termination states.*

Detailed formalization of the simulation relation and proof for the theorem can be found in Appendix B.3-B.4, where we focus on the proof for major passes C_{op} and C_{ptr} and omit details for the optional pass C_{callee} whose functional correctness is relatively more straightforward.

Public Noninterference. Next, we briefly justify that the transformed program satisfies software public noninterference. The detailed proof can be found in Appendix B.5. In our transformation, we pick the address shift δ so that $|\delta|$ is larger than the input program's maximum stack size. Then, we can denote the original stack region as $s_{\text{pub}} = [sp_{\text{init}} + \delta, sp_{\text{init}})$ and the new secret stack region as $s_{\text{sec}} = [sp_{\text{init}} + 2\delta, sp_{\text{init}} + \delta)$, where sp_{init} is the stack base. Intuitively, C will apply either `TransOp` or `TransPtr` to ensure that every instruction that accesses the original stack s_{pub} and operates on tainted data will have its target address shifted by δ and access the secret stack s_{sec} instead. Hence, C successfully ensures that the transformed program never stores secrets to the public stack region, thereby satisfying software public noninterference.

8 EVALUATION

We start by elaborating on the implementation of *SecSep*, the associated hardware defense, and the experiment setup for evaluation. We then present the evaluation results to answer the following three questions:

(1) How much performance overhead is introduced by the transformation of *SecSep* compared to the existing work, ProSpeCT [18]? (Section 8.2)
(2) How much annotation effort is required by *SecSep*?(Section 8.3)
(3) How efficient is *SecSep* toolchain to infer, check, and transform programs? (Appendix D)

8.1 Implementation and Experiment Setup

***SecSep* Toolchain.** We implement a prototype toolchain in OCaml, using Z3 [19] as the SMT solver. It includes (1) a parser to read *SecSep* annotations from the C source code, (2) an assembly parser to read x86-64 machine code from the compiled program, (3) a type inference implementation (Section 6), (4) a checker that verifies whether inferred types satisfy all the typing rules (Section 5.2),

and (5) a tool transforming assembly program using inferred types. This prototype only covers the instructions in the benchmarks used for evaluation. The toolchain incorporates LLVM/Clang to compile both the original and the transformed benchmark.

Hardware Defense. We implement our hardware-defense part in gem5 simulator v22.1 [9, 38] replicating the defense idea from ProSpeCT [18]. Specifically, modules including ROB, register file, scheduler (`InstructionQueue`), load/store queue, and branch squash logic (`Commit`, `IEW`) are modified to implement taint tracking and mechanisms to delay transmitter instructions leaking secrets. We apply a microarchitecture configuration similar to that used in prior work on secure speculation [15]. We model an 8-issue out-of-order superscalar processor with 32 load-queue entries, 32 store-queue entries, and 192 ROB entries. We use a tournament branch-prediction policy with 4096 BTB entries and 16 RAS entries. The memory system models a 32 KB 4-way L1 I-cache, a 64 KB 8-way L1 D-cache, and a 2 MB 16-way L2 cache, with 64 B cache lines.

Experiment Setup. We evaluate *SecSep* on six cryptographic benchmarks: our own implementation of `salsa20`, and five other benchmarks – `sha512`, `chacha20`, `poly1305`, `x25519` and `ed25519_sign` – from BoringSSL [24]. `ed25519_verify` is excluded due to the current lack of declassification support, which can be implemented with minor extensions (see Section 9). To minimize the instability due to cold caches, each benchmark is modified to repeat its main routine 100 times. In addition, we apply slight changes to some of the benchmarks, the details of which can be found in Appendix C.

We conduct our experiments on a test platform equipped with an Intel® Core™ i9-14900K CPU. Benchmarks are transformed using $\delta = 8$MB and run in gem5 under syscall emulation mode.

8.2 Performance of Transformed Programs

We evaluate and compare the performance overhead of the following four transformation schemes:

(1) ProSpeCT (public stack): Manually annotate secret function variables, relocate them to a global region while treating the original stack as public. Note that this scheme is insecure as annotation at the source-code level does not protect secrets spilled to the stack by compilers.
(2) ProSpeCT (secret stack): Manually annotate public function variables and perform the same relocation as scheme (1) while treating the original stack as secret. This approach conservatively protects any register spills and thus is secure.
(3) *SecSep* (no C_{callee}): After inferring *Octal* types, perform transformation using passes of C_{op} and C_{ptr} (Section 7).
(4) *SecSep*: Perform all the three passes, including C_{op}, C_{ptr}, and the C_{callee} optimization.

Software Overhead. We compare the execution time of transformed programs running on unmodified hardware, scaled to the execution time of the original program before transformation, shown in Figure 12(a). The goal is to focus on understanding the software overhead introduced by the additional or transformed instructions and their microarchitectural impacts.

On average, all the schemes have relatively low overhead below 4.2%, with ProSpeCT (public stack) having the highest overhead and *SecSep* (no C_{callee}) having a close-to-zero overhead. The performance difference mainly comes from the number of extra instructions introduced during transformation and whether the transformed program accesses regions far from the stack, which can result in worse cache performance.

Comprehensive Overhead. We now examine the execution time of the transformed programs on hardware equipped with security defenses, shown in Figure 12(b). The goal is to understand how different transformations influence overtraining due to imprecise memory partitioning, as well as the combined overhead introduced by both software and hardware.

On average, *SecSep* achieves the lowest overhead of 1.2% among all schemes, while ProSpeCT (secret stack) can incur as high as 151.1% overhead. Compared to other secure schemes, *SecSep* benefits from a more precise memory partition, thereby minimizing overtainting and enables efficient execution when the hardware defense is enabled. ProSpeCT (public stack) can also achieve relatively low overhead. However, its performance gains stem from undertainting, which compromises security. Nevertheless, it remains slower than *SecSep*, likely due to the notable software overhead illustrated in 12(a).

Effect of C_{callee}. To assess the role of C_{callee}, we compare *SecSep* (no C_{callee}) with *SecSep*. When C_{callee} is absent, the software overhead is lowered by 0.9%. However when hardware defense is enabled, a significant overhead increase of 25.1% is observed. This highlights the severity of overtainting caused by called routines if they are not handled properly. Therefore, despite adding extra instructions, C_{callee} is essential for efficient and secure speculation. Given its effectiveness, we include it as a key component in *SecSep*'s standard transformation scheme.

8.3 Manual Effort

In this section, we evaluate and compare the manual effort required by ProSpeCT and *SecSep* to annotate the source code for transformation. We focus on ProSpeCT's public stack scheme, as it exhibits efficient execution and can be manually patched to address its insecurity [18]. Metrics are shown in Table 1. For ProSpeCT, local variables in all functions (denoted as **F**) need to be examined, while *SecSep* only requires its user to examine arguments of functions that are in the binary's call graph (denoted as **F'**). To enable a fair comparison, we also count the number of local variables and ProSpeCT annotations required when examining only **F'**, with these numbers enclosed in parentheses.

As revealed in Table 1, the number of *SecSep* annotations grows linearly with the number of function arguments. This is because, to use *SecSep*, it suffices to identify the memory layout and taint types of all function arguments for functions in **F'**. An annotation burst is incurred at `poly1305` due to the frequent use of a 17-field structure as function arguments, while 14 of them share identical attributes and thus identical annotation. Writing *SecSep* annotations takes low effort, given that cryptographic functions usually have clear interfaces and seldom use complex data structures with unpredictable sizes (e.g. linked lists).

We also observe that, the number of variables to examine using ProSpeCT (**#Var**) is generally higher than the number of function arguments to examine using *SecSep* (**#Arg**), even when the scope is restrict to **F'** when counting **#Var**. This indicates that *SecSep* places less burden on the user, by requesting only function-interface-level

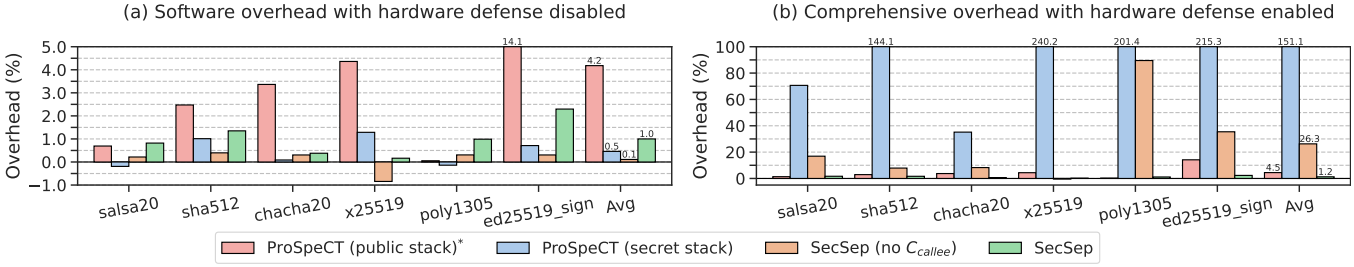


Figure 12: Execution time overhead of transformed programs relative to original programs. * means the scheme is not secure.

Benchmark				ProSpeCT (pub stack)		*SecSep*	
Name	LOC	#F	#F'	#Var	#Anno	#Arg	#Anno
salsa20	72	5	3	9 (5)	4 (2)	6	5
sha512	290	16	3	24 (2)	14 (1)	6	5
chacha20	100	7	3	8 (7)	4 (3)	8	7
x25519	1034	41	5	361 (11)	335 (4)	10	7
poly1305	314	11	7	33 (32)	26 (25)	11	74
ed25519_sign	2314	72	11	512 (77)	476 (69)	25	24

Table 1: Comparison of annotation efforts between ProSpeCT and *SecSep*. #F denotes the total number of functions present in the benchmark, while #F' denotes the number of functions called during the execution of the benchmark. #Var denotes the number of function local variables that need to be examined for correct annotation in ProSpeCT, and #Arg denotes the number of function arguments that need to be examined for correct annotation in *SecSep*. #Anno reports the number of lines of annotation in the corresponding technique.

annotations to harden cryptographic programs.

9 LIMITATIONS AND FUTURE WORK

While *SecSep* offers appealing features to rewrite assembly programs and separate secret/public data automatically, we acknowledge that it has several limitations worth discussing. First, we use heuristic-based type inference to transform assembly programs compiled by the off-the-shelf compiler LLVM, where the heuristics are developed through an empirical review of these assembly programs. The limitation is that it is not guaranteed to handle all possible assembly code patterns, and we need new heuristics for new compiler optimizations. This limitation can be alleviated by more engineering efforts to derive better heuristics.

Second, our inference algorithm relies on extra information (e.g., memory layout, valid regions, and taint) provided by source-code annotations to generate types. On the one hand, it is relatively straightforward for the programmer to provide these annotations since they only need to emphasize the high-level meanings of function arguments. On the other hand, we acknowledge that extra manual effort is required to go through each function argument, thereby increasing the barrier to using our tools (evaluated in Section 8.3). Future work might be conducted to improve the inference algorithm to release the need for these manual annotations.

Third, we provided a prototype to demonstrate the overall idea, while more features could be supported to improve the usability of our tool. For example, declassification is an essential notion in cryptographic programs supported by prior works such as ProSpeCT [18] while not included in our type system. To extend our prototype to support declassification, one can define a special function that takes a secret input and writes it to a given address in the public region (passed as a parameter of the function). The function is excluded from type inference and checking, so the program can call this function to write secrets to public regions for declassification. *SecSep* can also be improved to be compatible with more programs by supporting dynamically linked libraries and handling analysis with dynamically allocated heap data and pointer type casting.

10 RELATED WORK

We first discuss prior works on typed assembly language to justify the novelty and contribution of *Octal*. Next, we discuss prior mitigations, including software and hardware approaches, that aim to protect cryptographic programs against speculative-execution attacks. We also discuss prior work that transforms programs to separate secret and public data via a compiler approach.

Typed assembly language. Prior works [17, 23, 26, 41, 42] propose typed assembly language (TAL) and type-preserving compilation from high-level programs to TAL, where the types help guarantee the security of assembly programs. Instead of compiling high-level programs to generate assembly programs, *SecSep* rewrites assembly programs generated by an off-the-shelf compiler (LLVM) while using inferred types to guarantee security and functional correctness. Hence, the transformed programs still benefit from the optimizations of realistic compilers.

Jiang et al. [33] also propose a type system and corresponding type-inference algorithm for assembly programs. Their type system introduces more accurate information-flow tracking at bit granularity and helps detect side-channel vulnerabilities in cryptographic libraries. Note that the soundness of their type system relies on an assumption of memory safety, while our type system accurately tracks possible address ranges of each memory access and guarantees memory safety.

Other prior works [6, 8, 53] design information-flow type systems to guarantee that well-typed programs satisfy speculative constant time and implement their approaches in the Jasmin framework [3]. In this framework, the developers directly program in Jasmin, an assembly-like programming language, which requires

more manual effort compared with programming in higher-level languages such as C and is not compatible with some off-the-shelf cryptographic libraries such as BoringSSL [24].

Software mitigations against Spectre. Several prior works [6, 8, 14, 43, 44, 46, 52–54, 64] harden cryptographic programs against Spectre attacks by analyzing the programs' speculative control flow and blocking insecure speculation at the software level, e.g., by memory-fence insertion or speculative load hardening (SLH) [11]. Many of these approaches [14, 43, 44, 46, 54, 64] introduce large performance overhead since they unavoidably block safe speculation conservatively when blocking insecure speculation. Other works [6, 8, 52, 53] managed to achieve speculative noninterference with marginal overhead by applying SLH smartly, but they are built upon research-prototype source languages such as FaCT [13] or Jasmin [3, 4], thereby not compatible with off-the-shelf cryptographic libraries such as BoringSSL [24]. Furthermore, many of them [8, 14, 46, 52–54, 64] only consider speculation at conditional branches in their speculative control-flow analysis, so they cannot prevent leakage introduced by other speculation primitives [12, 29, 32, 35, 36, 48].

Hardware mitigations against Spectre. Prior works [7, 15, 37, 56, 62, 63] adopt hardware taint tracking and delay speculative operations that transmit the secret. These pure hardware solutions require no software changes but face challenges in identifying secret data in memory. STT [63] and others [7, 56, 62] only consider speculatively loaded data as secrets, while leaving nonspeculatively loaded data unprotected, thereby not guaranteeing constant time [31]. SPT [15] solves this problem by considering all data loaded from memory as tainted and only marking data as public if it is transmitted by the program nonspeculatively, but it introduces complex hardware changes to achieve good performance.

The authors of ProSpeCT [18] and others [22, 49] propose to make the program separate secret and public data into coarse-grained regions so that the hardware can easily identify secrets. This paper complements their solution by providing an automatic approach to generate a program satisfying this requirement.

Compiler approach to separating secret and public data. ConfLLVM [10] also uses a transformation technique that separates secret/public stack data into different regions. However, its static analysis cannot guarantee a program never writes secret data to the public region, so it relies on inserted run-time checks to ensure the correctness of the separation, which introduces extra overhead.

11 CONCLUSION

This paper proposed *Octal*, a new variant typed assembly language that helps rewrite cryptographic programs so that they split their secret and public data across coarse-grained memory regions. We provide a heuristic inference algorithm to infer the types of off-the-shelf cryptographic programs and automate the transformation process. The transformed programs enable hardware with fine-grained taint tracking at the register level and coarse-grained taint tracking at the memory level to achieve secure speculation with low performance overhead.

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*. Springer, 225–242.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1807–1823.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 965–982.

[5] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 623–639.

[6] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. 2025. Protecting Cryptographic Code Against Spectre-RSB:(and, in Fact, All Known Spectre Variants). In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 933–948.

[7] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 151–164.

[8] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-assurance cryptography in the Spectre era. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1884–1901.

[9] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[10] Ajay Brahmakshatriya, Piyus Kedia, Derrick P McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. 2019. ConfLLVM: A compiler for enforcing data confidentiality in low-level code. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[11] Chandler Carruth. n.d.. *Speculative Load Hardening*. https://llvm.org/docs/SpeculativeLoadHardening.html

[12] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical foundations for software Spectre defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 666–680.

[13] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 174–189.

[14] Rutvik Choudhary, Alan Wang, Zirui Neil Zhao, Adam Morrison, and Christopher W Fletcher. 2023. Declassiflow: A static analysis for modeling non-speculative knowledge to relax speculative execution security measures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2053–2067.

[15] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 607–622.

[16] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE symposium on security and privacy*. IEEE, 45–60.

[17] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software* (Atlanta, GA, USA). 25–35.

[18] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7161–7178.

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[20] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 843–857.

[21] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[22] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. Spectreguard: An efficient data-centric defense mechanism against Spectre attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[23] Neal Glew and Greg Morrisett. 1999. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 250–261.

[24] Google. n.d.. *BoringSSL*. https://boringssl.googlesource.com/boringssl

[25] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 955–972.

[26] Dan Grossman and Greg Morrisett. 2000. Scalable certification for typed assembly language. In *International Workshop on Types in Compilation*. Springer, 117–145.

[27] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-channel analysis of cryptographic software via early-terminating multiplications. In *International Conference on Information Security and Cryptology*. Springer, 176–192.

[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 368–379.

[29] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1853–1869.

[30] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[31] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1868–1883.

[32] Jann Horn. 2018. Speculative execution, variant 4: Speculative store bypass. https://project-zero.issues.chromium.org/issues/42450580. (2018).

[33] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache refinement type for side-channel detection of cryptographic software. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1583–1597.

[34] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[35] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[36] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[37] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing speculation with the principle of transient non-observability. In *30th USENIX Security Symposium (USENIX Security 21)*. 1397–1414.

[38] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jer'onimo Castrill'on, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago M"uck, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and 'Eder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[39] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2109–2122.

[40] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* 47 (2019), 538–570.

[41] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 1998. Stack-based typed assembly language. In *International Workshop on Types in Compilation*. Springer, 28–52.

[42] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

[43] Nicholas Mosier, Hamed Nemati, John C Mitchell, and Caroline Trippel. 2024. Serberus: Protecting cryptographic code from spectres at compile-time. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4200–4219.

[44] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. 1433–1450.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 1–20.

[46] Marco Patrignani and Marco Guarnieri. 2021. Exorcising spectres with secure compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 445–461.

[47] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. 565–581.

[48] Hernán Ponce-de León and Johannes Kinder. 2022. Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 235–248.

[49] Michael Schwarz, Moritz Lipp, Claudio Alberto Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A generic approach for mitigating Spectre. In *Network and Distributed System Security Symposium 2020*.

[50] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 279–299.

[51] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 131–145.

[52] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre declassified: Reading from the right place at the wrong time. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1753–1770.

[53] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing high-speed cryptography against Spectre v1. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1094–1111.

[54] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with Blade. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–30.

[55] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[56] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 572–586.

[57] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[58] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[59] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 888–904.

[60] Yuval Yarom and Katrina Falkner. 2014. FLUSH + RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. 719–732.

[61] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7 (2017), 99–112.

[62] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 707–720.

[63] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[64] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7125–7142.

A TYPE SAFETY

A.1 Supplementary Notes on *Octal*

In this section, we explain some details about *Octal*'s typing rules omitted in the main sections, and provide extra assumptions on well-typed programs.

A.1.1 Input and Block Type Variables. In *Octal*, we call the type variables of each function's input block state type "input variables" (since they serve as the function's arguments) and other variables only used in each non-input block's state types "block variables." Input variables are shared among all state type contexts within the same function. Hence, Typing-Jne requires that for branch instruction $\textbf{jne}\ \ell^{\sigma}$, its branch annotation satisfies $\text{getInputVar}(\text{dom}(\sigma)) = \emptyset$, so that when σ is used to instantiate type variables in the target block's context, it only instantiates block variables and leave input variables unchanged.

Furthermore, *Octal* also expects that all taint variables are input variables and shared among all blocks in the same function, as reflected by $\text{getTaintVar}(\text{dom}(\sigma)) = \emptyset$ in Typing-Jne. This requirement is necessary to prove the functional correctness of our transformation.

A.1.2 Explanation of isSpill. In Section 5.2, we introduce a predicate isSpill to check whether a given memory slot is a stack spill and invoke different typing rules (e.g., Typing-StoreOp-Spill or Typing-StoreOp-Non-Spill) accordingly. Specifically, we rely on debug tables generated by Clang to retrieve this information.

However, there is one special case worth discussing. Typing rules in *Octal* (Typing-Callq and State-Subtype) require that when calling a callee function, each of its memory slots (including its local stack slots) must belong to one of the caller's memory slots. Since functions usually do not directly access their callee's stack , we expect each function to have a special slot (denoted as $s_{\text{calleeStack}}$) that marks the memory region reserved for its callee's local stack so that it satisfies the above constraint.

Note that $s_{\text{calleeStack}}$ is not a typical stack spill slot, but we still make $\text{isSpill}(s_{\text{calleeStack}}) = \mathit{true}$ for the simplicity of typing constraints. Recall that we impose uniformity on the taint type of other non-spill slots so that the unified taint can be used as a hint for transformation. On the other hand, $s_{\text{calleeStack}}$ is not used in the caller's scope, so we do not need to assign it a uniform taint for transformation purposes. Imposing $\text{isSpill}(s_{\text{calleeStack}}) = \mathit{true}$ is a trick to avoid this constraint while still making the type system sound and the transformation functionally correct.

A.1.3 Definition of updateMem. When calling a function, part of the caller's memory slots will be accessed and modified by the callee, including the memory slots passed to the callee through pointers, and the special slot $s_{\text{calleeStack}}$ reserved for the callee's local stack. Typing-Callq invokes a predicate updateMem to generate the caller's memory type after the function call, which is defined in Algorithm 1. Note that updateMem has several extra assertions which further constrain well-typed programs.

First, if a memory slot s is accessed by the callee, s must either be $s_{\text{calleeStack}}$, the slot reserved for the callee's local stack slot, or a non-spill slot. This constraint is reasonable since stack spill slots

are used to hold register values temporarily and do not refer to any specific data objects that might be passed to a callee function.

Second, all local stack slots of the callee (including its spill slots) should belong to $s_{\text{calleeStack}}$, and we expect them to be uninitialized (i.e., have empty valid regions) before and after the call.

Third, for ease of illustrating the proof idea, we assume that each memory slot (except for $s_{\text{calleeStack}}$) is updated by at most one slot of the callee's function, while the general case can be defined and proved similarly. Furthermore, the callee slot's train type should be the same as the caller slot's, so that the slot is relocated by the same offset in the caller and callee during transformation.

Algorithm 1 Update memory type after function call (updateMem)

Input: Caller's memory type before call $\mathcal{M}_0$, callee's memory type at return (represented under caller's type context) $\mathcal{M}_1$

$S \leftarrow \emptyset$
$S[s \mapsto \{\}]$ for each $s \in \text{dom}(\mathcal{M}_0)$
for $s_1 \in \text{dom}(\mathcal{M}_1)$ **do**
  Find $s \in \text{dom}(\mathcal{M}_0)$ such that $s_1 \subseteq s$
  $S[s \mapsto S[s] \cup \{s_1\}]$
end for
$\mathcal{M} \leftarrow \mathcal{M}_0$
for $s \in \text{dom}(\mathcal{M})$ **do**
  $(s_0^{\text{valid}}, (e_0, \tau_0)) \leftarrow \mathcal{M}_0[s]$
  if $S[s] = \emptyset$ **then**
    continue
  else if isSpill(s) **then** ▷ s should be $s_{\text{calleeStack}}$, the slot reserved for callee's local stack slots, so s and all slots in $S[s]$ should be uninitialized.
    assert $s_0^{\text{valid}} = \emptyset$
    for $s_1 \in S[s]$ **do**
      assert $\mathcal{M}_1[s_1] = (\emptyset, \_)$
    end for
  else
    assert $S[s] = \{s_1\}$ for some s_1 ▷ We only discuss the case where each slot is updated by one callee's slot
    $(s_1^{\text{valid}}, (e_1, \tau_1)) \leftarrow \mathcal{M}_1[s_1]$
    assert $\neg$isSpill(s_1) $\wedge$ $\tau_0 = \tau_1$ ▷ We expect s_1 is not a spill slot from the callee's view.
    $s^{\text{valid}} \leftarrow (s_0^{\text{valid}} \backslash s_1) \cup s_1^{\text{valid}}$
    if $s^{\text{valid}} = s_1^{\text{valid}}$ **then**
      $\mathcal{M}[s \mapsto (s^{\text{valid}}, (e_1, \tau_0))]$
    else
      assert isNonChangeExp(e_0) $\wedge$ isNonChangeExp(e_1)
      $\mathcal{M}[s \mapsto (s^{\text{valid}}, (\top, \tau_0))]$
    end if
  end if
end for

A.1.4 Assumptions on Well-typed Programs. We provide extra assumptions on well-typed programs by constraining function types in Figure 13. Specifically, rule Typing-Func states that for a well-typed function:

- The main function directly halts, and other functions only return at the f_{ret} block.

$$\textsc{Typing-Prog}\quad \frac{\forall f \in \text{dom}(P).\ P, \mathcal{P} \vdash P(f) : \mathcal{P}(f)}{\vdash P : \mathcal{P}}$$

$$\textsc{Typing-Func}\quad \frac{\begin{array}{c} P(f) = F \qquad \forall \ell \in \text{dom}(F).\ \mathcal{P}, \Gamma \vdash F(\ell) : \Gamma(\ell) \\ \forall \ell \in \text{dom}(F), \ell \neq f_{\text{ret}}.\ \textbf{halt}, \textbf{retq} \notin F(\ell) \\ F(f_{\text{ret}}) = (f = \mathit{main})\ ?\ \textbf{halt} : \textbf{retq} \qquad \Gamma(f) = (\Delta_0, \mathcal{R}_0, \mathcal{M}_0) \\ \Gamma(f_{\text{ret}}) = (\Delta_1, \mathcal{R}_1, \mathcal{M}_1) \qquad \mathcal{R}_0[r_{\texttt{rsp}}] = \mathcal{R}_1[r_{\texttt{rsp}}] = (sp, 0) \\ \forall s \in \text{dom}(\mathcal{M}_{0,1}).\ \Delta_{0,1} \vdash s \cap [sp, sp+8) = \emptyset \\ \forall s, \text{isLocalStack}(s).\ \mathcal{M}_0[s] = (\emptyset, \_) \wedge \mathcal{M}_1[s] = (\emptyset, \_) \\ \forall r, \text{isCalleeSaved}(r).\ (\exists x \neq \top.\ \mathcal{R}_0[r] = \mathcal{R}_1[r] = (x, \_)) \\ \forall s, \mathcal{M}_1[s] = (s^{\text{valid}}, \beta).\ s^{\text{valid}} = \emptyset \vee \text{isNonChangeExp}(\beta) \\ \forall r \in \text{dom}(\mathcal{R}_1), \neg\text{isCalleeSaved}(r).\ \text{isNonChangeExp}(\mathcal{R}_1[r]) \\ \forall \ell \in \text{dom}(F), (\Delta, \mathcal{R}, \mathcal{M}) = \Gamma(\ell).\ \Delta_0 \subseteq \Delta \end{array}}{P, \mathcal{P} \vdash F : \Gamma}$$

Figure 13: Program typing

- The dependent type of `rsp` at the beginning of the function is always represented by a special type variable *sp*.
- The memory slot used to store the return address does not belong to the function's memory type, which implies that the function will never overwrite the return address during its lifetime.
- Memory slots on the function's local stack are marked as invalid (i.e., uninitialized) at the beginning and end of the function.
- Callee-saved registers (including `rsp`) are restored to their initial values when entering the function before return.
- Values of caller-saved registers and memory slots that are valid in the return state should not depend on pointer values, so they will not be affected by our transformation on pointers.
- Type constraints in the type context at the beginning of the function must also be constrained by the type context at other basic blocks.

Furthermore, we assume that each function only jumps to other functions through function calls and returns.

A.1.5 Memory Configuration. For ease of illustration, we provide a fixed configuration for public/secret and stack/non-stack memory regions as follows:

- Public stack: $s_{\text{stackPub}} = [sp_{\text{init}} + \delta, sp_{\text{init}})$
- Secret stack: $s_{\text{stackSec}} = [sp_{\text{init}} + 2\delta, sp_{\text{init}} + \delta)$
- Public non-stack: s_{otherPub}
- Secret non-stack: s_{otherSec}

With this configuration, we provide extra constraints on the memory layout of the well-typed program (to be transformed later): First, the program should only put its stack in s_{stackPub} and keep s_{stackSec} untouched so that our transformation can relocate data objects to this region without overwriting other valuable data. Second, as we focus on separating secret/public data in the stack region, we expect the original program to separate non-stack data properly, so we require that all data in s_{otherPub} be untainted.

A.2 *Octal*'s Abstract Machine

In Section 5.1, we defined *Octal*'s abstract machine state as (R, M, pc), which only covers the register file, memory, and the current PC. For the convenience of describing the relation between states and state types, we extend the state to (R, M, Φ), where Φ records PC and type substitution to instantiate type variables for the current state and all call sites on the call stack using constants:

S	::=	(R, M, Φ)	*State*
R	::=	$\{r_1 : (v_1, t_1), \ldots, r_n : (v_n, t_n)\}$	*Register File*
M	::=	$\{addr_1 : (v_1, t_1), \ldots, addr_n : (v_n, t_n)\}$	*Memory*
Φ	::=	$[]\|(pc, (\sigma_f, \sigma_b)); \Phi$	*PC/type substitution stack*
σ	::=	$\overrightarrow{x} \to \overrightarrow{e}$	*Type subsitution*

Specifically, σ_f is used to instantiate type variables in state types of all instructions in the current function, and σ_b is for type variables in state types of all instructions in the current block.

We then define the operational semantics for this extended abstract machine in Figure 14 and constrain well-formedness in Figure 15. Specifically, we use the helper function getStateType to denote the operation of applying typing rules in Figure 3 and deriving the state type at each instruction from the state type of its basic block (recorded in $\mathcal{P}$). We also define the helper function getDom as

$$\text{getDom}(\mathcal{M}, \sigma) = \bigcup_{s \in \text{dom}(\mathcal{M})} \sigma(s).$$

A.3 Type Safety Proof

Theorem 5 (Type Safety). *If $P, \mathcal{P} \vdash_{TAL} S$, then there exists S_1 such that $S \to S_1$ and $P, \mathcal{P} \vdash_{TAL} S_1$, or S is a termination state.*

Proof. Denote $S = (R, M, (pc, (\sigma_f, \sigma_b)); \Phi)$. According to Well-Formed-State-Base and Well-Formed-State-Inductive, pc is a valid PC in P. Let $(f, inst) = \text{getFuncInst}(P, pc)$. We also denote $(\Delta, \mathcal{R}, \mathcal{M}) = \text{getStateType}(\mathcal{P}, pc)$ as the state type for S, and $\sigma = \sigma_f \cup \sigma_b$ as the full type substitute to instantiate all type variables in the state type using constants. Consider the following cases for *inst*:

Case movq $r_1, i_d(r_b, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}$. First, according to Typing-Movq-r-m, registers r_b, r_i are untainted, and we can denote their type as $\mathcal{R}[r_b] = (e_b, 0), \mathcal{R}[r_i] = (e_i, 0)$. Then, according to Reg-Type, there should be $R[r_b] = (v_b, 0)$ and $R[r_i] = (v_i, 0)$, where $v_b = \sigma(e_b)$ and $v_i = \sigma(e_i)$. We further denote $\mathcal{R}[r_1] = (e_r, \tau_r)$ and $R[r_1] = (v_r, t_r)$, where $v_r = \sigma(e_r)$ and $t_r = \sigma(\tau_r)$. According to both Typing-StoreOp-Spill and Typing-StoreOp-Non-Spill, there should always be $\Delta \vdash \tau_r \Rightarrow \tau_{\text{op}}$, so $t_r \Rightarrow \sigma(\tau_{\text{op}})$. Therefore, S satisfies constraints in Dyn-Movq-r-m and is allowed to execute *inst*.

Second, we construct the next state S_1 as follows:

$$e_a = i_d + e_b + e_i \times i_s \quad v_a = \sigma(e_a) = i_d + v_b + v_i \times i_s + i_d$$
$$M_1 = M[[v, v+8) \mapsto (v_r, t_r)] \quad pc_1 = \text{nextPc}(P, pc)$$
$$S_1 = (R, M_1, (\text{nextPc}(P, pc), (\sigma_f, \sigma_b)); \Phi).$$

Third, we prove that S_1 is well-formed, i.e., $P, \mathcal{P} \vdash_{\text{TAL}} S_1$. According to Typing-Movq-r-m, there should exists s_o^{valid}, e_o, and τ_o such that

$$\Delta, \mathcal{R}, \mathcal{M} \vdash \text{store}(i_d(r_b, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}, 8, \mathcal{R}[r_1]) : (s_o^{\text{valid}}, (e_o, \tau_o)),$$

Dyn-Movq-m-r

$$\frac{\begin{array}{c}\text{getInst}(P, pc) = \textbf{movq}\ i_d(r_b, r_i, i_s)^{s,\tau}, r \\ pc' = \text{nextPc}(P, pc) \quad R[r_b] = (v_b, 0) \quad R[r_i] = (v_i, 0) \\ e = i_d + v_b + v_i \times i_s \quad M[e, e+8] = (v_m, t_m) \\ t_m \Rightarrow (\sigma_f \cup \sigma_b)(\tau) \quad R' = R[r \mapsto (v_m, t_m)]\end{array}}{P \vdash (R, M, (pc, (\sigma_f, \sigma_b)); \Phi) \to (R', M, (pc', (\sigma_f, \sigma_b)); \Phi)}$$

Dyn-Movq-r-m

$$\frac{\begin{array}{c}\text{getInst}(P, pc) = \textbf{movq}\ r, i_d(r_b, r_i, i_s)^{s,\tau} \\ pc' = \text{nextPc}(P, pc) \quad R[r_b] = (v_b, 0) \\ R[r_i] = (v_i, 0) \quad e = i_d + v_b + v_i \times i_s \quad R[r] = (v_r, t_r) \\ t_r \Rightarrow (\sigma_f \cup \sigma_b)(\tau) \quad M' = M[[e, e+8) \mapsto (v_r, t_r)]\end{array}}{P \vdash (R, M, (pc, (\sigma_f, \sigma_b)); \Phi) \to (R, M', (pc', (\sigma_f, \sigma_b)); \Phi)}$$

Dyn-Jne

$$\frac{\begin{array}{c}\text{getInst}(P, pc) = \textbf{jne}\ \ell^{\sigma} \quad \text{getFunc}(P, pc) = \text{getFunc}(P, pc') \\ R[\texttt{ZF}] = (b, 0) \quad pc' = b\ ?\ \text{nextPc}(P, pc) : \text{getBlockPc}(P, \ell) \\ \sigma_b' = b\ ?\ \sigma_b : (\sigma_f \cup \sigma_b) \circ \sigma\end{array}}{P \vdash (R, M, (pc, (\sigma_f, \sigma_b)); \Phi) \to (R, M, (pc', (\sigma_f, \sigma_b')); \Phi)}$$

Dyn-Callq

$$\frac{\begin{array}{c}\text{getInst}(P, pc) = \textbf{callq}\ f^{\sigma_{\text{call}}, \sigma_{\text{ret}}} \quad \text{getBlockPc}(P, f) = pc' \\ R[r_{\texttt{rsp}}] = (v, 0) \quad R' = R[r_{\texttt{rsp}} \mapsto (v-8, 0)] \\ M' = M[[v-8, v) \mapsto (\text{nextPc}(P, pc), 0)] \\ \Phi' = (pc', ((\sigma_f \cup \sigma_b) \circ \sigma_{\text{call}}, [] \to [])); (pc, (\sigma_f, \sigma_b)); \Phi\end{array}}{P \vdash (R, M, (pc, (\sigma_f, \sigma_b)); \Phi) \to (R', M', \Phi')}$$

Dyn-Retq

$$\frac{\begin{array}{c}\text{getInst}(P, pc) = \textbf{retq} \\ \text{getInst}(P, pc_p) = \textbf{callq}\ f^{\sigma_{\text{call}}, \sigma_{\text{ret}}} \quad R[r_{\texttt{rsp}}] = (v, 0) \\ R' = R[r_{\texttt{rsp}} \mapsto (v+8, 0)] \quad pc' = \text{nextPc}(P, pc_p) \\ M[v, v+8] = (pc', 0) \quad \Phi' = (pc', (\sigma_{f_p}, (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p})); \Phi\end{array}}{P \vdash (R, M, (pc, (\sigma_f, \sigma_b)); (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi) \to (R', M, \Phi')}$$

Figure 14: Dynamic operational semantics for ISA machine with taint tracking and type substitution

and for $\mathcal{M}_1 := \mathcal{M}[s_{\text{op}} \mapsto (s_o^{\text{valid}}, (e_o, \tau_o))]$, there should be $(\Delta, \mathcal{R}, \mathcal{M}_1) = \text{getStateType}(\mathcal{P}, pc_1)$.

We aim to prove that $\vdash (R, M_1, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M}_1)$. According to Reg-Mem-Type, as all other requirements can be easily derived by unfolding $\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})$ (implied by $P, \mathcal{P} \vdash_{\text{TAL}} S$), we just need to prove $\vdash M_1 : \sigma(\mathcal{M}_1)$.

For each $s \in \text{dom}(\mathcal{M}_1)$, denote $\mathcal{M}_1[s] = (s_1^{\text{valid}}, (e_1, \tau_1))$. According to Mem-Type, we need to prove the following statements

$$\begin{array}{l}(s \subseteq s_{\text{stackPub}} \vee s \subseteq s_{\text{otherPub}} \vee s \subseteq s_{\text{otherSec}}) \\ (\text{isSpill}(s) \Rightarrow s \subseteq s_{\text{stackPub}}) \quad (s \subseteq s_{\text{otherPub}} \Rightarrow \tau_1 = 0) \\ (s_1^{\text{valid}} \subseteq s) \quad (\vdash M : \mathcal{M}[s])\end{array} \tag{1}$$

If $s \neq s_{\text{op}}$, then $\mathcal{M}_1[s] = \mathcal{M}[s]$ and $\Delta \vdash s \cap s_{\text{op}} = \emptyset$ since all memory slots are non-overlapped. According to both Typing-StoreOp-Spill and Typing-StoreOp-Non-Spill, there should be $\Delta \vdash [e_a, e_a+8) \subseteq s_{\text{op}}$. Thus, there should be $\Delta \vdash [e_a, e_a+8) \cap s = \emptyset$, so $[v_a, v_a +$

$$\textsc{Value-Type}\ \frac{v = e \vee e = \top \qquad t \Rightarrow \tau}{\vdash (v,t) : (e,\tau)} \qquad \textsc{Reg-Type}\ \frac{\forall r \in \mathrm{dom}(\mathcal{R}).\ \vdash R[r] : \mathcal{R}[r]}{\vdash R : \mathcal{R}}$$

$$\textsc{Mem-Slot-Range-Type}\ \frac{\vdash M[v_1, v_2] : \beta}{\vdash M : ([v_1, v_2), \beta)} \qquad \textsc{Mem-Slot-Empty-Type}\ \frac{}{\vdash M : (\emptyset, \beta)}$$

$$\textsc{Mem-Slot-Top-Type}\ \frac{\forall x \in s^{\mathrm{valid}}.\ \vdash M[x] : (\top, \tau)}{\vdash M : (s^{\mathrm{valid}}, (\top, \tau))}$$

$$\textsc{Mem-Type}\ \frac{\begin{array}{c}\forall s, \mathcal{M}[s] = (s^{\mathrm{valid}}, (\_, \tau)).\\ ((s \subseteq s_{\mathrm{stackPub}} \vee s \subseteq s_{\mathrm{otherPub}} \vee s \subseteq s_{\mathrm{otherSec}})\\ \wedge(\mathrm{isSpill}(s) \Rightarrow s \subseteq s_{\mathrm{stackPub}}) \wedge (s \subseteq s_{\mathrm{otherPub}} \Rightarrow \tau = 0)\\ \wedge(s^{\mathrm{valid}} \subseteq s) \wedge (\vdash M : \mathcal{M}[s]))\end{array}}{\vdash M : \mathcal{M}}$$

$$\textsc{Reg-Mem-Type}\ \frac{\begin{array}{c}\mathrm{getInputVar}(\mathrm{dom}(\sigma_f)) = \mathrm{dom}(\sigma_f)\\ \mathrm{getInputVar}(\mathrm{dom}(\sigma_b)) = \emptyset\\ \sigma = \sigma_f \cup \sigma_b \qquad \sigma(\Delta) = \{true\} \qquad \vdash R : \sigma(\mathcal{R})\\ \vdash M : \sigma(\mathcal{M}) \qquad \forall s \in \mathrm{dom}(\mathcal{M}).\ \mathrm{getVars}(s) = \sigma_f(s)\end{array}}{\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})}$$

$$\textsc{Well-Formed-State-Base}\ \frac{\vdash P : \mathcal{P} \qquad \mathrm{getFunc}(P, pc) = main \qquad \vdash (R, M, (\sigma_f, \sigma_b)) : \mathrm{getStateType}(\mathcal{P}, pc)}{P, \mathcal{P} \vdash_{\mathrm{TAL}} (R, M, (pc, (\sigma_f, \sigma_b)))}$$

$$\textsc{Well-Formed-State-Inductive}\ \frac{\begin{array}{c}\vdash P : \mathcal{P}\\ \mathrm{getFunc}(P, pc) = f \neq main \qquad (\Delta, \mathcal{R}, \mathcal{M}) = \mathrm{getStateType}(\mathcal{P}, pc)\\ \vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})\\ s_{\mathrm{ret}} = [\sigma_f(sp), \sigma_f(sp) + 8) \qquad M[s_{\mathrm{ret}}] = (\mathrm{nextPc}(P, pc_p), 0)\\ \forall x \notin \mathrm{getDom}(\mathcal{M}, \sigma_f) \cup s_{\mathrm{ret}}.\ M[x] = M_p[x]\\ \mathrm{getInst}(P, pc_p) = \mathbf{callq}\ f^{\sigma_{\mathrm{call}}, \sigma_{\mathrm{ret}}} \qquad \sigma_f = (\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\mathrm{call}}\\ P, \mathcal{P} \vdash_{\mathrm{TAL}} (R_p, M_p, (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi)\end{array}}{P, \mathcal{P} \vdash_{\mathrm{TAL}} (R, M, (pc, (\sigma_f, \sigma_b)); (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi)}$$

Figure 15: Well-formed machine states, where *sp* represents `rsp`'s dependent type at the beginning of a function and appears in σ_f

$8) \cap \sigma(s) = \emptyset$. This implies that for all $x \in \sigma(s)$, $M_1[x] = M[x]$. Therefore, (1) holds.

If $s = s_{\mathrm{op}}$, then $s_1^{\mathrm{valid}} = s_o^{\mathrm{valid}}$, $e_1 = e_o$, $\tau_1 = \tau_o$, and we need to derive s_o^{valid}, e_o, τ_o to prove (1). Denote $\mathcal{M}[s_{\mathrm{op}}] = (s_{o0}^{\mathrm{valid}}, (\_, \tau_{o0}))$. We consider the following cases:

- $\mathrm{isSpill}(s_{\mathrm{op}})$: According to Tying-StoreOp-Spill, there should be $s_o^{\mathrm{valid}} = [e_a, e_a + 8)$, $e_o = e_r$, and $\tau_o = \tau_{\mathrm{op}}$.
- $\neg\mathrm{isSpill}(s_{\mathrm{op}})$ and $s_{o0}^{\mathrm{valid}} \subseteq [e_a, e_a + 8)$: According to Typing-StoreOp-Non-Spill, there should be $s_o^{\mathrm{valid}} = [e_a, e_a + 8)$, $e_o = e_r$, and $\tau_o = \tau_{\mathrm{op}}$.
- $\neg\mathrm{isSpill}(s_{\mathrm{op}})$ and $s_{o0}^{\mathrm{valid}} \not\subseteq [e_a, e_a + 8)$: According to Typing-StoreOp-Non-Spill, there should be $s_o^{\mathrm{valid}} = [e_a, e_a + 8) \cup s_{o0}^{\mathrm{valid}}$, $e_o = \top$, and $\tau_o = \tau_{\mathrm{op}}$.

According to the following discussion, we can know that if $s_{\mathrm{op}} \subseteq s_{\mathrm{otherPub}}$, then $\tau_o = 0$:

- $\mathrm{isSpill}(s_{\mathrm{op}})$: according to Mem-Type and $P, \mathcal{P} \vdash_{\mathrm{TAL}} S$, we know that $\mathrm{isSpill}(s_{\mathrm{op}}) \Rightarrow s_{\mathrm{op}} \subseteq s_{\mathrm{stackPub}}$. So $s_{\mathrm{op}} \not\subseteq s_{\mathrm{otherPub}}$ and we can ignore this case.
- $\neg\mathrm{isSpill}(s_{\mathrm{op}})$: According to Typing-StoreOp-Spill, $\tau_{o0} = \tau_{\mathrm{op}} = \tau_o$. Furthermore, by unfolding $P, \mathcal{P} \vdash_{\mathrm{TAL}} S$ we can know that $s_{\mathrm{op}} \subseteq s_{\mathrm{otherPub}} \Rightarrow \tau_{o0} = 0$. Therefore, $\tau_o = 0$.

Furthermore, other statements in (1) can be proved by unfolding $P, \mathcal{P} \vdash_{\mathrm{TAL}} S$ and performing a discussion on whether $\mathrm{isSpill}(s_{\mathrm{op}})$ is true or not. Therefore, $\vdash (R, M_1, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M}_1)$.

If $f := \mathrm{getFunc}(P, pc) = main$, Well-Formed-State-Base implies that $P, \mathcal{P} \vdash_{\mathrm{TAL}} S_1$.

We consider the case where $f \neq main$. There should exists pc_p, σ_{f_p}, σ_{b_p}, and Φ_0 such that $\Phi = (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0$. Furthermore, $P, \mathcal{P} \vdash_{\mathrm{TAL}} S$ implies that there exists R_p, M_p such that $P, \mathcal{P} \vdash_{\mathrm{TAL}} (R_p, M_p, \Phi)$.

Note that $[v_a, v_a + 8) \subseteq \sigma(s_{\mathrm{op}}) \subseteq \mathrm{getDom}(\mathcal{M}, \sigma_f)$, and $s_{\mathrm{ret}} \cap \mathrm{getDom}(\mathcal{M}, \sigma_f) = \emptyset$ (implied by Lemma 7), and $M_1[x] = M[x]$ for all $x \notin [v_a, v_a + 8)$, there should be

$$M_1[s_{\mathrm{ret}}] = M[s_{\mathrm{ret}}] = (\mathrm{nextPc}(P, pc_p), 0)$$
$$\forall x \notin \mathrm{getDom}(\mathcal{M}, \sigma_f) \cup s_{\mathrm{ret}}.\ M_1[x] = M[x] = M_p[x].$$

Therefore, the statement also holds for the case when $f \neq main$.

Case jne $\ell^{\sigma_{\mathrm{op}}}$. First, according to Typing-Jne, flag `ZF` is untainted and we can denote its type as $\mathcal{R}[\mathtt{ZF}] = (e = 0, 0)$. Then, according to Reg-Type, Value-Type, and well-formedness of S, there should be $R[\mathtt{ZF}] = (v, 0)$ where $v = \sigma(e = 0)$. Thus, S satisfies constraints in Dyn-Jne and is allowed to execute *inst*.

Second, we construct the next state S_1 as follows:

$$S_1 = \begin{cases}(R, M, (\mathrm{nextPc}(P, pc), (\sigma_f, \sigma_b); \Phi) & v = true\\ (R, M, (\mathrm{getBlockPc}(P, \ell), (\sigma_f, (\sigma_f \cup \sigma_b) \circ \sigma_{\mathrm{op}}); \Phi) & v = false.\end{cases}$$

Third, we prove that S_1 is well-formed under both cases (when the branch is taken and not taken). When $v = true$, i.e., the branch is not taken, denote the next PC as $pc_1 = \mathrm{nextPc}(P, pc)$. According to Typing-Jne, $\mathrm{getStateType}(\mathcal{P}, pc_1) = (\Delta \cup \{e = 0\}, \mathcal{R}, \mathcal{M})$. Note that $\sigma(e = 0) = v = true$ and $\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})$, so there should also be $\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta \cup \{e = 0\}, \mathcal{R}, \mathcal{M})$. Thus, we can derive that $P, \mathcal{P} \vdash_{\mathrm{TAL}} S_1$ according to Well-Formed-State-Base and Well-Formed-State-Inductive.

When $v = false$, i.e., the branch is taken, denote next PC as $pc_1 = \mathrm{getBlockPc}(P, \ell)$. Similarly to the not-taken case, we can prove that

$$\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}). \tag{2}$$

According to Typing-Jne, we can also know that

$$(\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}) \sqsubseteq \sigma_{\mathrm{op}}(\mathcal{P}(f)(\ell)). \tag{3}$$

Since pc_1 is the PC of block ℓ in function f, the state type for pc_1 should be $\text{getStateType}(\mathcal{P}, pc_1) = \mathcal{P}(f)(\ell)$. We aim to prove that $\vdash (R, M, (\sigma_f, (\sigma_f \cup \sigma_b) \circ \sigma_{\text{op}})) : \mathcal{P}(f)(\ell)$. Denote $(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) = \mathcal{P}(f)(\ell)$ and $\sigma_1 = \sigma_f \cup ((\sigma_f \cup \sigma_b) \circ \sigma_{\text{op}})$. According to Reg-Mem-Type, we just need to prove the following statements:

- $\sigma_1(\Delta_1) = \mathit{true}$: According to Lemma 9, $\sigma_1(\Delta_1) = ((\sigma_f \cup \sigma_b) \circ \sigma_{\text{op}})(\Delta_1) = (\sigma_f \cup \sigma_b)(\sigma_{\text{op}}(\Delta_1))$. According to (3) and State-Subtype, $\Delta \cup \{e \neq 0\} \vdash \sigma_{\text{op}}(\Delta_1)$. According to (2) and Lemma 8, we can know that $(\sigma_f \cup \sigma_b)(\sigma_{\text{op}}(\Delta_1)) = \mathit{true}$. Thus, the statement is true.
- $\vdash R : \sigma_1(\mathcal{R}_1)$: For each $r \in \text{dom}(\mathcal{R}_1)$, denote $R[r] = (v, t)$ and $\mathcal{R}_1[r] = (e_1, \tau_1)$. According to (3) and Reg-Subtype, $r \in \mathcal{R}[r]$, and when denoting $\mathcal{R}[r] = (e, \tau)$, there should be

$$\Delta \vdash e = \sigma_{\text{op}}(e_1) \vee (\text{isNonChangeExp}(e) \wedge \sigma_{\text{op}}(e_1) = \top)$$
$$\Delta \vdash \tau \Rightarrow \sigma_{\text{op}}(\tau_1).$$

  According to (2) and Lemma 8, the above statements still hold when we instantiate their type variables with $\sigma = (\sigma_f \cup \sigma_b)$ (and also simplify with Lemma 9), so we can get

$$\sigma(e) = \sigma_1(e_1) \vee (\text{isNonChangeExp})(\sigma(e)) \wedge \sigma_1(e_1) = \top)$$
$$\sigma(\tau) \Rightarrow \sigma_1(\tau).$$

  Furthermore, (2) also implies that

$$v = \sigma(e) \vee \sigma(e) = \top \qquad t \Rightarrow \sigma(\tau).$$

  By combining the above statements and performing a simple case discussion on whether $\sigma_1(e_1) = \top$, we can finally get

$$v = \sigma_1(e_1) \vee \sigma_1(e_1) = \top \qquad t \Rightarrow \sigma_1(\tau_1).$$

  Thus, the statement is true.
- $\vdash M : \sigma_1(\mathcal{M}_1)$: For each $s_1 \in \text{dom}(\mathcal{M}_1)$, denote $\mathcal{M}_1[s_1] = (s_1^{\text{valid}}, (e_1, \tau_1))$. According to (3) and State-Subtype, there exists $s \in \mathcal{M}$ such that

$$\Delta \vdash \sigma_{\text{op}}(s_1) \subseteq s \wedge \mathcal{M}[s] \sqsubseteq \sigma_{\text{op}}(\mathcal{M}_1[s_1]).$$

  Denote $\mathcal{M}[s] = (s^{\text{valid}}, (e, \tau))$. Then, according to Mem-Slot-Subtype, there should be

$$\Delta \vdash \sigma_{\text{op}}(s_1^{\text{valid}}) \subseteq s^{\text{valid}}$$
$$\Delta \vdash \text{isSpill}(\sigma_{\text{op}}(s_1^{\text{valid}})) \Rightarrow \text{isSpill}(s^{\text{valid}})$$
$$\Delta \vdash e = \sigma_{\text{op}}(e_1) \vee (\text{isNonChangeExp}(e) \wedge \sigma_{\text{op}}(e_1) = \top) \vee \sigma_{\text{op}}(s_1^{\text{valid}}) = \emptyset$$
$$\Delta \vdash \tau = \sigma_{\text{op}}(\tau_1) \vee (\text{isSpill}(s^{\text{valid}}) \wedge \sigma_{\text{op}}(s_1^{\text{valid}}) = \emptyset).$$

  According to (2), the above statements still hold when we instantiate their type variables with $\sigma = (\sigma_f \cup \sigma_b)$ (and also simplify with Lemma 9), so we can get

$$\sigma_1(s_1) \subseteq \sigma(s)$$
$$\sigma_1(s_1^{\text{valid}}) \subseteq \sigma(s^{\text{valid}})$$
$$\text{isSpill}(\sigma_1(s_1^{\text{valid}})) \Rightarrow \sigma(s^{\text{valid}})$$
$$\sigma(e) = \sigma_1(e_1) \vee (\text{isNonChangeExp}(\sigma(e)) \wedge \sigma_1(e_1) = \top) \vee \sigma_1(s_1^{\text{valid}}) = \emptyset$$
$$\sigma(\tau) = \sigma_1(\tau_1) \vee (\text{isSpill}(\sigma(s^{\text{valid}})) \wedge \sigma_1(s_1^{\text{valid}}) = \emptyset).$$

  (2) implies that $\vdash M : \sigma(\mathcal{M})$, so there should be

$$\sigma(s) \subseteq s_{\text{stackPub}} \vee \sigma(s) \subseteq s_{\text{otherPub}} \vee \sigma(s) \subseteq s_{\text{otherSec}}$$
$$\text{isSpill}(\sigma(s)) \Rightarrow \sigma(s) \subseteq s_{\text{stackPub}}$$
$$\sigma(s) \subseteq s_{\text{otherPub}} \Rightarrow \sigma(\tau) = 0 \qquad \sigma(s^{\text{valid}}) \subseteq s$$
$$\vdash M : \sigma(\mathcal{M}[s]).$$

  By combining the above statements and performing a simple discussion on $\sigma_1(s_1^{\text{valid}})$ and $\sigma_1(\mathcal{M}_1[s_1])$, we can know that the statement is true.
- $\forall s \in \text{dom}(\mathcal{M}_1).\ \text{getVars}(s) = \text{dom}(\sigma_f)$: According to Lemma 6, $\text{dom}(\mathcal{M}_1) = \text{dom}(\mathcal{M})$. According to (2), for each s in $\text{dom}(\mathcal{M})$, $\text{getVars}(s) = \text{dom}(\sigma_f)$. Thus, the statement is true.

So far, we have successfully proved that $\vdash (R, M, (\sigma_f, (\sigma_f \cup \sigma_b) \circ \sigma_{\text{op}})) : \mathcal{P}(f)(\ell)$. Therefore, it is straightforward to derive that $P, \mathcal{P} \vdash_{\text{TAL}} S_1$ from this statement and $P, \mathcal{P} \vdash_{\text{TAL}} S$.

Case callq $f_c^{\sigma_{\text{call}}, \sigma_{\text{ret}}}$. First, according to Typing-Callq, $r_{\texttt{rsp}}$ is untainted, and we can denote its type as $\mathcal{R}[r_{\texttt{rsp}}] = (e, 0)$. Then, according to Reg-Type, Value-Type, and well-formedness of S, there should be $R[r_{\texttt{rsp}}] = (v, 0)$ where $v = \sigma(e)$. Thus, S satisfies constraints in Dyn-Callq and is allowed to execute *inst*.

Second, we construct the next state S_1 as follows:

$$pc_c = \text{getBlockPc}(P, f_c)$$
$$R_1 = R[r_{\texttt{rsp}} \mapsto (v - 8, 0)]$$
$$M_1 = M[[v - 8, v) \mapsto (\text{nextPc}(P, pc), 0)]$$
$$\Phi_1 = (pc_c, (\sigma \circ \sigma_{\text{call}}, [] \to [])); (pc, (\sigma_f, \sigma_b)); \Phi$$
$$S_1 = (R_1, M_1, \Phi_1).$$

Furthermore, let $\mathcal{R}_1 = \mathcal{R}[r_{\texttt{rsp}} \mapsto (e - 8, 0)]$. $P, \mathcal{P} \vdash_{\text{TAL}} S$ implies $\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})$. Thus, we can derive $\vdash (R_1, M_1, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}_1, \mathcal{M})$ by proving constraints in Reg-Mem-Type. Specifically, the constraint $\vdash M_1 : \sigma(\mathcal{M})$ can be implied from $\vdash M : \sigma(\mathcal{M})$, $\mathcal{M}[e - 8, e] = (\emptyset, \_)$, and $M_1[x] = M[x]$ for all $x \notin [v - 8, v]$. According to Typing-Callq, there should also be $(\Delta, \mathcal{R}_1, \mathcal{M}) \sqsubseteq \sigma_{\text{call}}(\mathcal{P}(f_c)(f_c))$.

Third, we prove that S_1 is well-formed. Since pc_c is the PC of f_c, the state type for pc_c should be $\text{getStateType}(\mathcal{P}, pc_c) = \mathcal{P}(f_c)(f_c)$. We aim to prove that $\vdash (R_1, M_1, (\sigma \circ \sigma_{\text{call}}, [] \to [])) : \mathcal{P}(f_c)(f_c)$. Denote $\sigma_c = \sigma \circ \sigma_{\text{call}}$ and $(\Delta_c, \mathcal{R}_c, \mathcal{M}_c) = \mathcal{P}(f_c)(f_c)$. According to Reg-Mem-Type, we just need to prove the following statements:

- $\sigma_c(\Delta_c) = \mathit{true}$, $\vdash R_1 : \sigma_c(\mathcal{R}_c)$, and $\vdash M_1 : \sigma_c(\mathcal{M}_c)$: Note that $\vdash (R_1, M_1, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}_1, \mathcal{M})$ and $(\Delta, \mathcal{R}_1, \mathcal{M}) \sqsubseteq \sigma_{\text{call}}(\mathcal{P}(f_c)(f_c))$, we can prove the three statements following a similar strategy used to prove for the case of **jne**.
- $\forall s \in \text{dom}(\mathcal{M}_c).\ \text{getVars}(s) = \text{dom}(\sigma_c)$: this is true since σ_{call} is a substitute that instantiate all variables in $\mathcal{P}(f_c)(f_c)$.

Therefore, $\vdash (R_1, M_1, (\sigma \circ \sigma_{\text{call}}, [] \to [])) : \mathcal{P}(f_c)(f_c)$.

Let $s_{\text{retc}} = [\sigma_c(sp), \sigma_c(sp) + 8)$. According to $(\Delta, \mathcal{R}_1, \mathcal{M}) \sqsubseteq \sigma_{\text{call}}(\mathcal{P}(f_c)(f_c))$ and Reg-Subtype, there should be $e - 8 = \sigma_{\text{call}}(sp)$, so $\sigma_c(sp) = \sigma(\sigma_{\text{call}}(sp)) = \sigma(e - 8) = v - 8$. Thus, we have $M_1[s_{\text{retc}}] = M_1[v - 8, v] = (\text{nextPc}(P, pc), 0)$.

To prove $P, \mathcal{P} \vdash_{\text{TAL}} S_1$, we still need to prove the following statements (here we intend to use R and M to prove the existence of R_p and M_p in Well-Formed-State-Inductive):

- $\forall x \notin \text{getDom}(\mathcal{M}_c, \sigma_c) \cup s_{\text{retc}}.\ M_1[x] = M[x]$: this hold since $M_1[x] = M[x]$ for all $x \notin s_{\text{retc}}$.
- $\text{getInst}(P, pc) = \textbf{callq}\ f_c^{\sigma_{\text{call}}, \sigma_{\text{ret}}}$: this is our assumption.
- $\sigma_c = (\sigma_f \cup \sigma_b) \circ \sigma_{\text{call}}$: this is the definition of σ_c.
- $P, \mathcal{P} \vdash_{\text{TAL}} (R, M, (pc, (\sigma_f, \sigma_b)); \Phi$: this is our assumption.

Case retq. First, according to Typing-Func, $f \neq main$. Then, there should exists $pc_p, \sigma_{f_p}, \sigma_{b_p}$ and Φ_0 such that $\Phi = (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0$.

According to Typing-Func, the state type of pc is $(\Delta, \mathcal{R}, \mathcal{M}) := \text{getStateType}(f)(f_{\text{ret}})$. It also implies that $\mathcal{R}[r_{\texttt{rsp}}] = (sp, 0)$. Then, according to Reg-Type, Value-Type, and well-formedness of S, there should be $R[r_{\texttt{rsp}}] = (v, 0)$, where $v = \sigma(sp) = \sigma_f(sp)$. Furthermore, according to $P, \mathcal{P} \vdash_{\text{TAL}} S$ and Well-Formed-State-Inductive, there should exist R_p and M_p such that

$$
\begin{aligned}
&s_{\text{ret}} = [\sigma_f(sp), \sigma(sp) + 8) = [v, v + 8) \\
&M[s_{\text{ret}}] = (\text{nextPc}(P, pc_p), 0) \\
&\text{getInst}(P, pc_p) = \textbf{callq}\ f^{\sigma_{\text{call}}, \sigma_{\text{ret}}} \\
&\forall x \notin \text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}.\ M[x] = M_p[x] \\
&\sigma_f = (\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}} \\
&P, \mathcal{P} \vdash_{\text{TAL}} (R_p, M_p, \Phi).
\end{aligned}
$$

Thus, S satisfies all constraints in Dyn-Retq and is allowed to execute *inst*.

Second, we construct the next state S_1 as follows:

$$
\begin{aligned}
&R_1 = R[r_{\texttt{rsp}} \mapsto (v + 8, 0)] \\
&\Phi_1 = (\text{nextPc}(P, pc_p), (\sigma_{f_p}, (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}); \Phi_0 \\
&S_1 = (R_1, M, \Phi_1).
\end{aligned}
$$

Third, we prove that S_1 is well-formed. Let

$$
\begin{aligned}
&(\Delta_0, \mathcal{R}_0, \mathcal{M}_0) = \text{getStateType}(\mathcal{P}, pc_p) \\
&(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(\Delta, \mathcal{R}, \mathcal{M}) \\
&(\Delta_2, \mathcal{R}_2, \mathcal{M}_2) = (\Delta_0 \cup \Delta_1, \mathcal{R}_1[r_{\texttt{rsp}} \mapsto \mathcal{R}_0[r_{\texttt{rsp}}]], \\
&\qquad\qquad \text{updateMem}(\mathcal{M}_0, \mathcal{M}_1)).
\end{aligned}
$$

According to Well-Formed-State-Inductive, there should be $P, \mathcal{P} \vdash_{\text{TAL}} (R_p, M_p, \Phi)$, so

$$\vdash (R_p, M_p, (\sigma_{f_p}, \sigma_{b_p})) : (\Delta_0, \mathcal{R}_0, \mathcal{M}_0). \tag{4}$$

According to rule Typing-Callq, there should be $(\Delta_2, \mathcal{R}_2, \mathcal{M}_2) = \text{getStateType}(\mathcal{P}, \text{nextPc}(P, pc_p))$. We aim to prove

$$\vdash (R_1, M, (\sigma_{f_p}, (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p})) : (\Delta_2, \mathcal{R}_2, \mathcal{M}_2).$$

Denote $\sigma_1 = \sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}$. According to Reg-Mem-Type, we just need to prove the following:

- $\sigma_1(\Delta_2) = \{true\}$: For each statement e in Δ_2, there should be $e \in \Delta_0$ or $e \in \Delta_1$. If $e \in \Delta_0$, then $\sigma_1(e) = (\sigma_{f_p} \cup \sigma_{b_p})(e) = true$ (implied by (4)). If $e \in \Delta_1$, then there exists $e' \in \Delta$ such that $e = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(e')$ and $(\sigma_f \cup \sigma_b)(e') = true$. According to Well-Formed-State-Inductive, we have $\sigma_f = (\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}}$. Then, according to Lemma 10,

$$
\begin{aligned}
\sigma_1(e) &= (\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p})(e) \\
&= ((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}) \circ (\sigma_{\text{call}} \cup \sigma_{\text{ret}}))(e') \\
&= (((\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}}) \cup \sigma_b)(e') \\
&= (\sigma_f \cup \sigma_b)(e') = true.
\end{aligned}
$$

  Therefore, the statement holds.
- $\vdash R_1[r_{\texttt{rsp}}] : \sigma_1(\mathcal{R}_2[r_{\texttt{rsp}}])$: According to the definition of $(\Delta_0, \mathcal{R}_0, \mathcal{M}_0)$ and Typing-Callq, there should exists e_0 such that $\mathcal{R}_0[r_{\texttt{rsp}}] = (e_0, 0)$. Let $\mathcal{R}_{p0} = \mathcal{R}_0[r_{\texttt{rsp}} \mapsto (e - 8, 0)]$, we can also get $(\Delta_0, \mathcal{R}_{p0}, \mathcal{M}_0) \sqsubseteq \sigma_{\text{call}}(\mathcal{P}(f)(f))$. Thus, we can derive that $e_0 - 8 = \sigma_{\text{call}}(sp)$. Then, according to Lemma 10,

$$
\begin{aligned}
\sigma_1(\mathcal{R}_2[r]) &= (\sigma_1(\sigma_{\text{call}}(sp + 8)), 0) \\
&= ((\sigma_1 \circ (\sigma_{\text{call}} \cup \sigma_{\text{ret}}))(sp + 8), 0) \\
&= (\sigma(sp + 8), 0) = (v + 8, 0) = R_1[r_{\texttt{rsp}}].
\end{aligned}
$$

  Therefore, the statement holds.
- $\forall r \in \text{dom}(\mathcal{R}_2), r \neq r_{\texttt{rsp}}.\ \vdash R'[r] : \sigma_1(\mathcal{R}_2[r])$: According to Lemma 10, we have $\sigma_1(\mathcal{R}_2[r]) = (\sigma_1 \circ (\sigma_{\text{call}} \cup \sigma_{\text{ret}}))(\mathcal{R}[r]) = (\sigma_f \cup \sigma_b)(\mathcal{R}[r])$ and $R'[r] = R[r]$. Furthermore, we can derive $\vdash R[r] : (\sigma_f \cup \sigma_b)(\mathcal{R}[r])$ from the well-formedness of S. Thus, the original statement holds.
- $\vdash M : \sigma_1(\mathcal{M}_2)$: For each $s \in \text{dom}(\mathcal{M}_2)$, denote $\mathcal{M}_2[s] = (s_2^{\text{valid}}, (e_2, \tau_2))$. According to Mem-Type, as other requirements of well-formedness are straightforward to prove, we focusing on proving that $\vdash M : \sigma_1(\mathcal{M}_2[s])$.
  According to Algorithm 1, when executing updateMem to generate $\mathcal{M}_2$, we generate a map S that maps each slot in $\text{dom}(\mathcal{M}_2)$ to slots in $\text{dom}(\mathcal{M}_1)$ that belong to s. Consider the following cases:
  - $s = \sigma_{\text{call}}([sp, sp+8)) = [sp+c-8, sp+c)$ (where $\sigma_{\text{call}}(sp) = sp+c-8$ and $\mathcal{R}_2[r_{\texttt{rsp}}] = \mathcal{R}_0[r_{\texttt{rsp}}] = (sp+c, \_)$): according to Typing-Callq, $\mathcal{M}_0[s] = (\emptyset, \_)$. According to Typing-Func, for all $s_1 \in \text{dom}(\mathcal{M}_1)$, $s_1 \cap s = \emptyset$, so $S[s] = \emptyset$. These two facts imply that $s_2^{\text{valid}} = \emptyset$, so $\sigma_1(s_2^{\text{valid}}) = \emptyset$ and the statement holds according to Mem-Slot-Empty-Type.
  - $S[s] = \emptyset$ and $s \neq [sp + c - 8, sp + c)$: in this case, we have $s_2^{\text{valid}} = s_0^{\text{valid}}$ and $e_2 = e_0$. According to the well-formedness of R_p, M_p, there should be $\vdash M_p : \sigma_1(\mathcal{M}_0[s])$. Recall that $\forall x \notin \text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}.\ M[x] = M_p[x]$. By applying Lemma 15, it is also straightforward to derive that $\sigma_1(s_2^{\text{valid}}) \cap (\text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}) = \emptyset$, so $M[\sigma_1(s_2^{\text{valid}})] = M_p[\sigma_1(s_0^{\text{valid}})]$. Thus, we can derive $\vdash M : \sigma_1(\mathcal{M}_2[s])$ from $\vdash M_p : \sigma_1(\mathcal{M}_0[s])$.
  - $\text{cardinal}(S[s]) > 1$: according to updateMem, $s_2^{\text{valid}} = s_1^{\text{valid}} = \emptyset$, so $\sigma_1(s_2^{\text{valid}}) = \emptyset$ and the statement holds according to Mem-Slot-Empty-Type.
  - $\text{cardinal}(S[s]) = 1$: denote $S[s] = \{s_1\}$ and $\mathcal{M}_1[s_1] = (s_1^{\text{valid}}, (e_1, \tau_1))$. Then, $s_2^{\text{valid}} = (s_0^{\text{valid}} \backslash s_1) \cup s_1^{\text{valid}}$ and $\tau_2 = \tau_1$ according to updateMem. According to the definition of $\mathcal{M}_1$, there exists $s_c \in \mathcal{M}$ such that $s_c = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(s_1)$. Then, we have $\mathcal{M}_1[s_1] = (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(\mathcal{M}_c[s_c])$.
    By unfolding the assumption $P, \mathcal{P} \vdash_{\text{TAL}} S$, we can get $\vdash M : \sigma(\mathcal{M}[s_c])$. According to Lemma 10, for all e, $\sigma(e) =$

$\sigma_1((\sigma_{\text{call}} \cup \sigma_{\text{ret}})(e))$. These two facts imply that $\vdash M : \sigma_1(\mathcal{M}_1[s_1])$.
We then consider the following cases.
If $s_2^{\text{valid}} = s_1^{\text{valid}}$, then $e_2 = e_1$. We can derive $\vdash M : \sigma_1(\mathcal{M}_2[s])$ from $\vdash M : \sigma_1(\mathcal{M}_1[s_1])$.
If $s_2^{\text{valid}} \subsetneq s_1^{\text{valid}}$, then $e_2 = \top$. According to Mem-Slot-Top-Type, $\vdash M : \sigma_1(\mathcal{M}_2[s])$ holds.

□

Lemma 6. *For all $P, \mathcal{P}, pc_0, pc_1$, and for $i \in \{0,1\}$, $(\Delta_i, \mathcal{R}_i, \mathcal{M}_i) = getStateType(\mathcal{P}, pc_i)$, if $getFunc(P, pc_0) = getFunc(P, pc_1)$, then there should be $\text{dom}(\mathcal{M}_0) = \text{dom}(\mathcal{M}_1)$.*

Proof. This lemma means that memory state types of instructions in the same function have the same memory slots.

According to Typing-Func and Typing-Prog, all basic blocks in P are well-typed. By unfolding typing rules for each basic block, we know that $\text{dom}(\mathcal{M}_0) = \text{dom}(\mathcal{M}_1)$ holds if pc_0 and pc_1 are in the same basic block. According to Typing-Jne (and similar typing rules for other branch instructions omitted here), $\text{dom}(\mathcal{M}_0) = \text{dom}(\mathcal{M}_1)$ holds if pc_0 and pc_1 refer to a branch instruction and its target. Therefore, we can derive that $\text{dom}(\mathcal{M}_0) = \text{dom}(\mathcal{M}_1)$ holds as long as pc_0 and pc_1 are in the same function. □

Lemma 7. *For all $P, \mathcal{P}, pc$, and $(\Delta, \mathcal{R}, \mathcal{M}) = getStateType(\mathcal{P}, pc)$, there should be*

$$\forall s \in \text{dom}(\mathcal{M}).\ \Delta \vdash s \cap [sp, sp+8) = \emptyset.$$

Proof. Denote $f = \text{getFunc}(P, pc)$, $\ell = \text{getFunc}(P, pc)$, $(\Delta_0, \mathcal{R}_0, \mathcal{M}_0) = \mathcal{P}(f)(f)$, and $(\Delta_1, \_, \_) = \mathcal{P}(f)(\ell)$. Typing-Func and typing rules for instructions sequences implies that

$$\forall s \in \text{dom}(\mathcal{M}_0).\ \Delta_0 \vdash s \cap [sp, sp+8) = \emptyset,$$
$$\Delta_0 \subseteq \Delta_1, \qquad \Delta_1 = \Delta.$$

Furthermore, Lemma 6 implies that $\text{dom}(\mathcal{M}) = \text{dom}(\mathcal{M}_0)$. Therefore, the statement holds. □

Lemma 8. *Suppose $\vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M})$ where σ_f and σ_b are substitutes that instantiate type variables using constants. For any arithmetic statement e, if $\Delta \vdash e = true$, then $(\sigma_f \cup \sigma_b)(e) = true$.*

Proof. According to Reg-Mem-Type, $(\sigma_f \cup \sigma_b)(\Delta) = \{true\}$, which implies that $(\sigma_f \cup \sigma_b)$ is a valid substitute map that instantiates all type variables in Δ. Therefore, the statement holds. □

Lemma 9. *For all type substitute σ_f, σ_b, and σ_0, if $\text{dom}(\sigma_f) \cap \text{dom}(\sigma_b) = \emptyset$ and $\text{dom}(\sigma_f) \cap \text{dom}(\sigma_0) = \emptyset$, then for all e,*

$$(\sigma_f \cup ((\sigma_f \cup \sigma_b) \circ \sigma_0))(e) = ((\sigma_f \cup \sigma_b) \circ \sigma_0)(e).$$

Proof. First, we prove that the statement holds when $e = x$ is a type variable. If $x \in \text{dom}(\sigma_f)$, then $(\sigma_f \cup ((\sigma_f \cup \sigma_b) \circ \sigma_0))(x) = \sigma_f(x)$. Since $\text{dom}(\sigma_f) \cap \text{dom}(\sigma_0) = \emptyset$, then $x \notin \text{dom}(\sigma_0)$. In this case, applying σ_0 to instantiate x will still get x, i.e., $\sigma_0(x) = x$. Then,

$$((\sigma_f \cup \sigma_b) \circ \sigma_0)(x) = (\sigma_f \cup \sigma_b)(\sigma_0(x)) = (\sigma_f \cup \sigma_b)(x) = \sigma_f(x).$$

Thus, the statement holds for this case.

If $x \notin \text{dom}(\sigma_f)$, then

$$(\sigma_f \cup ((\sigma_f \cup \sigma_b) \circ \sigma_0))(e) = ((\sigma_f \cup \sigma_b) \circ \sigma_0)(e).$$

Therefore, the statement holds when e is a type variable.

For a general type expression e, the statement can be proved by applying the statement to each type variable in e. □

Lemma 10. *For all type substitute σ_{f_p}, σ_{b_p}, σ_b, σ_{call}, and σ_{ret}, if*

- *σ_{call} is a substitute that instnatiates each type variable in its domain using expressions over variables in $\text{dom}(\sigma_{f_p}) \cup \text{dom}(\sigma_{b_p})$.*
- *$\sigma_{ret} = \overrightarrow{x_1} \to \overrightarrow{x_2}$ is a substitute that instantiates each type variable in its domain using another unique type variable;*
- *$\text{dom}(\sigma_{ret}) = \text{dom}(\sigma_b)$ and $\text{dom}(\sigma_{ret}) \cap \text{dom}(\sigma_{call}) = \emptyset$;*
- *$\text{dom}(\sigma_{ret}^{-1}) \cap \text{dom}(\sigma_{f_p}) = \text{dom}(\sigma_{ret}^{-1}) \cap \text{dom}(\sigma_{b_p}) = \text{dom}(\sigma_{f_p}) \cap \text{dom}(\sigma_{b_p}) = \emptyset$*

then for all e,

$$((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{ret}^{-1}) \cup \sigma_{b_p}) \circ (\sigma_{call} \cup \sigma_{ret}))(e) = (((\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{call}) \cup \sigma_b)(e).$$

Proof. First, we prove that the statement holds when $e = x$ is a type variable. If $x \in \text{dom}(\sigma_{\text{call}})$, then $x \notin \sigma_b$ and $\sigma_{\text{call}}(x) \in \text{dom}(\sigma_{f_p}) \cup \text{dom}(\sigma_{b_p})$. Then, we have

$$\begin{aligned}
&((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}) \circ (\sigma_{\text{call}} \cup \sigma_{\text{ret}}))(x) \\
&= ((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}) \circ \sigma_{\text{call}})(x) \\
&= ((\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}})(x) \\
&= (((\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}}) \cup \sigma_b)(x).
\end{aligned}$$

If $x \in \text{dom}(\sigma_{\text{ret}})$, then $x \in \text{dom}(\sigma_b)$. Then, we have

$$\begin{aligned}
&((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}) \circ (\sigma_{\text{call}} \cup \sigma_{\text{ret}}))(x) \\
&= ((\sigma_{f_p} \cup (\sigma_b \circ \sigma_{\text{ret}}^{-1}) \cup \sigma_{b_p}) \circ \sigma_{\text{ret}})(x) \\
&= ((\sigma_b \circ \sigma_{\text{ret}}^{-1}) \circ \sigma_{\text{ret}})(x) = \sigma_b(x) \\
&= (((\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}}) \cup \sigma_b)(x).
\end{aligned}$$

Therefore, the statement holds when e is a type variable.

For a general type expression e, the statement can be proved by applying the statement to each type variable in e. □

B TRANSFORMATION

B.1 Base Transformation Pass

We begin with formalizing our base transformation pass C, which takes a well-typed program and generates a program that satisfies public noninterference. For this pass, besides well-formedness, we additionally assume that for all functions of the input program, pointer arguments are passed via registers for ease of illustration.

In Algorithm 2, we define how C transforms each instruction. In the following part, we will overload the same symbol to represent transforming instruction sequences or the whole program by applying C to each instruction (e.g., $C(P)$).

In this algorithm, the transformation strategy map ω_p is generated as described in Algorithm 3. The two passes C_{op} and C_{ptr} are formalized in Algorithm 4-5. When generating the transformation

strategy for each slot s, we additionally require that if s is not a local stack slot (i.e., both the current function and its caller will access it), and its transformation strategy is `TransOp` (i.e., $\omega(s) = \texttt{TransOp}$), then its corresponding taint type must be constant (either 0 or 1) (see the assertion in Algorithm 3). This restriction is crucial to guarantee functional correctness since the function and its caller must agree on where the slot should be relocated and whether to add δ to the memory operand when accessing the slot.

Algorithm 2 Transform Instruction (C)

Input: Instruction to transform *inst* and its *pc*, original program P and program type $\mathcal{P}$
Output: Transformed instructions ι

$F_p \leftarrow \text{getFunc}(P, pc)$
$(\Delta_p, \mathcal{R}_p, \mathcal{M}_p) \leftarrow \text{getStateType}(\mathcal{P}, pc)$
$\omega_p \leftarrow \text{getTransStrategy}(\mathcal{M}_p)$
switch *inst* **do**
  case **movq/addq/cmpq** op_1, op_0
    $\iota \leftarrow$ **movq/addq/cmpq** $C_{\text{op}}(\omega_p, op_1), C_{\text{op}}(\omega_p, op_0)$
  case **pushq/popq** r
    $\iota \leftarrow$ **pushsecq/popsecq** r
  case **callq** $f_c^{\sigma_{\text{call}}, \sigma_{\text{ret}}}$
    $\mathcal{S}_c \leftarrow \text{getStateType}(\mathcal{P}, \text{getBlockPc}(P, f_c))$
    $\iota_{\text{call}}, \sigma'_{\text{call}} \leftarrow C_{\text{ptr}}((\Delta_p, \mathcal{R}_p, \mathcal{M}_p), \sigma_{\text{call}}(\mathcal{S}_c), \sigma_{\text{call}})$
    $\iota \leftarrow \iota_{\text{call}}$; **callq** $f_c^{\sigma'_{\text{call}}, \sigma_{\text{ret}}}$
  default
    $\iota \leftarrow inst$
end switch

Algorithm 3 Determine Transformation Strategy

Input: Memory type $\mathcal{M}$
Output: Transformation strategy ω

$T \leftarrow \{\}$
▷ T is a map from each base pointers to a set of taint types of slots referenced by the pointer
for $s \in \text{dom}(\mathcal{M})$ **do**
  $ptr \leftarrow \text{getPtr}(s)$
  $(\_, (\_, \tau)) \leftarrow \mathcal{M}[s]$
  $T[ptr \mapsto T(ptr) \cup \{\tau\}]$
end for
$\omega \leftarrow \{\}$
for $s \in \text{dom}(\mathcal{M})$ **do**
  $ptr \leftarrow \text{getPtr}(s)$
  $(\_, (\_, \tau)) \leftarrow \mathcal{M}[s]$
  if $ptr \neq sp$ and $\text{cardinal}(T[ptr]) = 1$ **then**
    $\omega[s \mapsto \texttt{TransPtr}]$
  else
    $\omega[s \mapsto \texttt{TransOp}]$
    assert $\text{isLocalStack}(s) \vee \tau \in \{0, 1\}$
  end if
end for

Algorithm 4 Transform Operand (C_{op})

Input: Operand to transform *op*, transformation strategy map ω
Output: Transformed operand op'

$op' \leftarrow op$
switch *op* **do**
  case $i_d(r_b, r_i, i_s)^{s,\tau}$
    if $\omega(s) = \texttt{TransOp} \wedge \tau \neq 0$ **then**
      $op' \leftarrow \delta + i_d(r_b, r_i, i_s)^{s+\delta,\tau}$
    end if
end switch

Algorithm 5 Transform Pointer Arguments for Function Call (C_{ptr})

Input: State type at the call site $(\Delta_p, \mathcal{R}_p, \mathcal{M}_p)$, state type at the beginning of the callee function $(\Delta_c, \mathcal{R}_c, \mathcal{M}_c)$, original type substitution σ_{call} of the function call
Output: Instructions to transform pointer arguments ι, transformed type substitution σ'_{call}

$\omega_p \leftarrow \text{getTransStrategy}(\mathcal{M}_p)$
$\omega_c \leftarrow \text{getTransStrategy}(\mathcal{M}_c)$
$S_{\texttt{TransPtr}} \leftarrow \{\}$
for $s_c \in \mathcal{M}_c$ **do**
  Find $s_p \in \mathcal{M}_p$ such that $\sigma_{\text{call}}(s_c) \subseteq s_p$
  $(s_c^{\text{valid}}, (\_, \tau_c)) \leftarrow \mathcal{M}_c[s_c]$
  assert $\text{isSpill}(s_p) \Rightarrow s_c^{\text{valid}} = \emptyset$
  assert $\omega_c(s_c) = \texttt{TransOp} \Rightarrow \omega_p(s_p) = \texttt{TransOp}$
  if $\omega_c(s_c) = \texttt{TransPtr} \wedge \omega_p(s_p) = \texttt{TransOp} \wedge \sigma_{\text{call}}(\tau_c) \neq 0$ **then**
    ▷ If the pointer is not transformed in the caller, and the slot is tainted (needs to be relocated), then add δ to the pointer before passing to the callee
    assert $\neg\text{isNonChangeExp}(\text{getPtr}(s_c))$
    $S_{\texttt{TransPtr}} \leftarrow S_{\texttt{TransPtr}} \cup \{\text{getPtr}(s_c)\}$
  end if
end for
$\iota \leftarrow []$
$\sigma'_{\text{call}} \leftarrow \sigma_{\text{call}}$
for $ptr \in S_{\texttt{TransPtr}}$ **do**
  Find unique r such that $\mathcal{R}_c[r] = (ptr, 0)$
  $\iota \leftarrow$ **addq** $\delta, r; \iota$
  $\sigma'_{\text{call}}[ptr \mapsto \sigma'_{\text{call}}(ptr) + \delta]$
end for
assert $\forall r, \mathcal{R}_c[r] = (e, \_).\ e \in S_{\texttt{TransPtr}} \vee \sigma'_{\text{call}}(e) = \sigma_{\text{call}}(e)$
assert $\forall s, \mathcal{M}_c[s] = (\_, (e, \_)).\ \sigma'_{\text{call}}(e) = \sigma_{\text{call}}(e)$
▷ Sanity check: each transformed pointer argument is passed through a unique register and is independent from other registers and memory slots.

B.2 Optimization Pass

We then formalize our second transformation pass C_{callee}, which restores the callee-saved registers' taint after function calls. Intuitively, C_{callee} ensures that public callee-saved registers will not be conservatively marked as tainted after finishing the callee function, which prevents the hardware defense from introducing extra pipeline delays.

For the ease of implementing and describing this pass, we additionally assume that each function of the input program will use **pushq** to push all callee-saved registers it needs to change to the stack in the beginning, and restore them before returning to the call site. With this assumption, we can check the first block of the function to find in which slot the function saves the callee-saved registers. In Algorithm 6, we define how C_{callee} transforms each instruction (i.e., getSavedMemSlot(F_p, r)). Similarly, we can also use $C_{\text{callee}}(P)$ to represent the output program generated by applying C_{callee} to each instruction.

Algorithm 6 Restore Callee-saved Register Taint (C_{callee})

Input: Instruction to transform *inst* and its *pc*, original program P, and program type $\mathcal{P}$
Output: Transformed instructions ι

```
if inst = callq f_c^{σ_call,σ_ret} then
    F_p ← getFunc(P, pc)
    (Δ_p, R_p, M_p) ← getStateType(P, pc)
    ι_before, ι_after ← [], []
    Let S be the set of callee-saved registers used in F_p
    for r ∈ S do
        if R[r] = (_, 0) then                          ▷ r is untainted
            x ← getSavedMemSlot(F_p, r) − R_p[r_rsp]
            ι_before ← movq r, x(r_rsp); ι_before
            ι_after ← movq x(r_rsp), r; ι_after
        end if
    end for
    ι ← ι_before; inst; ι_after
else
    ι ← inst
end if
```

Note that we omit the discussion for proving functional correctness and public noninterference of the program generated by C_{callee}. The overall idea is to build the simulation relation between states of running $C(P)$ and $C_{\text{callee}}(C(P))$.

B.3 Simulation Relation

In this section, we define the simulation relation between states of running the original program and the transformed program $C(P)$ as shown in Figure 16.

At the high level, C relocates memory objects in memory to separate secret and public data and modifies pointer values accordingly. To justify functional correctness, we constrain the non-pointer values of registers and memory slots to be the same as those in the original state. Another key point of the simulation relation is to build a memory address (slot) map between the original and the transformed memory. Specifically, we define helper mapping functions used in the simulation relations as follows:

$$\text{getOpShift}(\mathcal{M}, \omega, s) = \begin{cases} 0 & \omega(s) = \texttt{TransPtr} \vee \mathcal{M}[s] = (\_, (\_, 0)) \\ \delta & \text{otherwise} \end{cases}$$

SIM-PC
$$\frac{\text{getFuncInstSeq}(P, pc) = (f, I) \quad \text{getFuncInstSeq}(C(P), pc') = (f, C(I))}{P, \mathcal{P} \vdash pc' \prec pc}$$

SIM-VAR-MAP
$$\frac{\text{dom}(\sigma) = \text{dom}(\sigma') \quad \forall x \in \text{dom}(\sigma).\ \text{isNonChangeExp}(x) \Rightarrow \sigma'(x) = \sigma(x)}{\Delta \vdash \sigma' \prec \sigma}$$

SIM-VAL
$$\frac{(e = \top \wedge v' = v) \vee (e \neq \top \wedge v' = \sigma'(e) \wedge v = \sigma(e))}{\vdash ((v', t), \sigma') \prec_{e,\tau} ((v, t), \sigma)}$$

SIM-REG
$$\frac{\forall r \in \mathcal{R}.\ (R'[r], \sigma') \prec_{\mathcal{R}[r]} (R[r], \sigma)}{\vdash (R', \sigma') \prec_{\mathcal{R}} (R, \sigma)}$$

SIM-MEM
$$\frac{\begin{array}{c}\omega = \text{getTransStrategy}(\mathcal{M}) \\ \forall s, \mathcal{M}[s] = (s^{\text{valid}}, (e, \tau)), x = \text{getPtr}(s). \\ ((\omega(s) = \texttt{TransOp} \Rightarrow \sigma(s) \subseteq s_{\text{stackPub}} \wedge \\ (\text{isLocalStack}(s) \vee \tau \in \{0, 1\})) \wedge \delta_{\text{op}} = \text{getOpShift}(\mathcal{M}, \omega, s) \wedge \\ \delta_{\text{ptr}} = \sigma'(x) - \sigma(x) \wedge \delta_{\text{ptr}} = \text{getPtrShift}(\mathcal{M}, \omega, \sigma, s) \wedge \\ (\sigma(s^{\text{valid}}) = \emptyset \vee (M'[\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op}}], \sigma') \prec_{e,\tau} (M[\sigma(s^{\text{valid}})], \sigma)))\end{array}}{\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)}$$

SIMULATION-RELATION-HELPER
$$\frac{\begin{array}{c}P, \mathcal{P} \vdash pc' \prec pc \qquad (\Delta, \mathcal{R}, \mathcal{M}) = \text{getStateType}(\mathcal{P}, pc) \\ P, \mathcal{P} \vdash (R, M, (\sigma_f, \sigma_b)) : (\Delta, \mathcal{R}, \mathcal{M}) \\ \Delta \vdash \sigma'_f \prec \sigma_f \qquad \Delta \vdash \sigma'_b \prec \sigma_b \qquad \sigma = \sigma_f \cup \sigma_b \\ \sigma' = \sigma'_f \cup \sigma'_b \qquad \vdash (R', \sigma') \prec_{\mathcal{R}} (R, \sigma) \qquad \vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)\end{array}}{P, \mathcal{P} \vdash (R', M', (pc', (\sigma'_f, \sigma'_b))) \prec_{\text{helper}} (R, M, (pc, (\sigma_f, \sigma_b)))}$$

SIMULATION-RELATION-BASE
$$\frac{\begin{array}{c}P, \mathcal{P} \vdash (R', M', (pc', (\sigma'_f, \sigma'_b))) \prec_{\text{helper}} (R, M, (pc, (\sigma_f, \sigma_b))) \\ P, \mathcal{P} \vdash_{\text{TAL}} (R, M, (pc, (\sigma_f, \sigma_b)))\end{array}}{P, \mathcal{P} \vdash (R', M', (pc', (\sigma'_f, \sigma'_b))) \prec (R, M, (pc, (\sigma_f, \sigma_b)))}$$

SIMULATION-RELATION-OTHER
$$\frac{\begin{array}{c}P, \mathcal{P} \vdash (R', M', (pc', (\sigma'_f, \sigma'_b))) \prec_{\text{helper}} (R, M, (pc, (\sigma_f, \sigma_b))) \\ P, \mathcal{P} \vdash_{\text{TAL}} (R, M, (pc, (\sigma_f, \sigma_b)); \Phi) \\ \Phi = (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0 \qquad \Phi' = (pc'_p, (\sigma'_{f_p}, \sigma'_{b_p})); \Phi'_0 \\ \text{getInst}(C(P), pc'_p) = \textbf{callq}\ f^{\sigma'_{\text{call}}, \sigma'_{\text{ret}}} \\ \sigma'_f = (\sigma'_{f_p} \cup \sigma'_{b_p}) \circ \sigma'_{\text{call}} \qquad pc'_{p0} = \text{getCallPrefixPc}(C(P), pc'_p) \\ P, \mathcal{P} \vdash (R'_p, M'_p, (pc'_{p0}, (\sigma'_{f_p}, \sigma'_{b_p})); \Phi'_0) \prec (R_p, M_p, (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0) \\ (\Delta, \mathcal{R}, \mathcal{M}) = \text{getStateType}(\mathcal{P}, pc) \\ \omega = \text{getTransStrategy}(P, pc) \\ s_{\text{ret}} = [\sigma_f(sp), \sigma_f(sp) + 8) \qquad M[s_{\text{ret}}] = (\text{nextPc}(P, pc_p), 0) \\ M'[s_{\text{ret}}] = (\text{nextPc}(C(P), pc'_p), 0) \\ \forall x \notin \text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}.\ M[x] = M_p[x] \\ \forall x \notin \text{getShiftedDom}(\mathcal{M}, \omega, \sigma_f, \sigma'_f) \cup s_{\text{ret}}.\ M'[x] = M'_p[x]\end{array}}{P, \mathcal{P} \vdash (R', M', (pc', (\sigma'_f, \sigma'_b)); \Phi') \prec (R, M, (pc, (\sigma_f, \sigma_b)); \Phi)}$$

Figure 16: Simulation Relation

$$\text{getPtrShift}(\mathcal{M},\omega,\sigma,s) = \begin{cases} 0 & \omega(s) = \texttt{TransOp} \vee \sigma(s) \notin s_{\text{stackPub}} \cup s_{\text{stackSec}} \\ \delta & \omega(s) = \texttt{TransPtr} \wedge \sigma(\mathcal{M}[s]) = (\_,(\_,1)) \\ 0,\delta & \omega(s) = \texttt{TransPtr} \wedge \sigma(\mathcal{M}[s]) = (\_,(\_,0)) \end{cases}$$

$$\text{getShift}(\mathcal{M},\omega,\sigma,\sigma',s) = \text{getOpShift}(\mathcal{M},\omega,s) + (\sigma'(\text{getPtr}(s)) - \sigma(\text{getPtr}(s))).$$

$$\text{getShiftedSlot}(\mathcal{M},\omega,\sigma,\sigma',s) = \sigma(s) + \text{getShift}(\mathcal{M},\omega,\sigma,\sigma',s)$$

$$\text{getPossibleSlot}(\mathcal{M},\omega,\sigma,\sigma',s) = \begin{cases} \sigma(s) \cup (\sigma(s)+\delta) & \omega(s) = \texttt{TransOp} \\ \sigma(s) + \sigma'(\text{getPtr}(s)) - \sigma(\text{getPtr}(s)) & \omega(s) = \texttt{TransPtr}. \end{cases}$$

$$\text{getDom}(\mathcal{M},\sigma) = \bigcup_{s\in\text{dom}(\mathcal{M})} \sigma(s)$$

$$\text{getShiftedDom}(\mathcal{M},\omega,\sigma,\sigma') = \bigcup_{s\in\text{dom}(\mathcal{M})} \text{getPossibleSlot}(\mathcal{M},\omega,\sigma,\sigma',s)$$

B.4 Functional Correctness

With the simulation defined in the last section, we formalize and prove the functional correctness of the transformed program generated by the base pass C as shown in Theorem 11.

Theorem 11. *If $P,\mathcal{P} \vdash_{TAL} S$ and $P,\mathcal{P} \vdash S' \prec S$, then $S \xrightarrow{inst} S_1$, $S' \xrightarrow{C(inst)}{}^{*} S'_1$, and $P,\mathcal{P} \vdash S'_1 \prec S_1$, or S and S' are termination states.*

Proof. Denote

$$S = (R, M, (pc, (\sigma_f,\sigma_b); \Phi)) \qquad \sigma = \sigma_f \cup \sigma_b$$
$$S' = (R', M', (pc', (\sigma'_f,\sigma'_b)); \Phi') \qquad \sigma' = \sigma'_f \cup \sigma_b.$$

According to the definitions of well-formedness and the simulation relation, pc and pc' are valid PCs in P and $C(P)$, respectively. Then, there should exist f, $inst$, I such that $\text{getFuncInstSeq}(P,pc) = (f, inst; I)$ and $\text{getFuncInstSeq}(C(P),pc') = (f, C(inst); C(I))$. We also denote

$$(\Delta,\mathcal{R},\mathcal{M}) = \text{getStateType}(\mathcal{P},pc) \qquad \omega = \text{getTransStrategy}(\mathcal{M}).$$

Consider the following cases for $inst$:

Case movq $r_1, i_d(r_b,r_i,i_s)^{s_{\text{op}},\tau_{\text{op}}}$. In this case, there should be $C(inst) =$ **movq** $r_1, \delta_0 + i_d(r_b,r_i,i_s)^{s_{\text{op}}+\delta_0,\tau_{\text{op}}}$ where if $\omega(s_{\text{op}}) = \texttt{TransOp} \wedge \tau_{\text{op}} \neq 0$, then $\delta_0 = \delta$, else $\delta_0 = 0$.

First, according to Typing-Movq-R-M, r_b and r_i are untainted and we can denote their type as $\mathcal{R}[r_b] = (e_b, 0)$, $\mathcal{R}[r_i] = (e_i, 0)$. Then, there should be

$$R[r_b] = (\sigma(e_b),0) \qquad R[r_i] = (\sigma(e_i),0)$$
$$R'[r_b] = (\sigma'(e_b),0) \qquad R'[r_i] = (\sigma'(e_i),0).$$

We further denote $\mathcal{R}[r_1] = (e_r,\tau_r)$, $R[r_1] = (v_r,t_r)$, and $R'[r_1] = (v'_r,t'_r)$, where $v_r = \sigma(e_r)$ and $v'_r = \sigma'(e_r)$. According to the simulation relation between S and S', there should be $t'_r = t_r$. According to both Typing-StoreOp-Spill and Typing-StoreOp-Non-Spill, there should always be $\Delta \vdash \tau_r \Rightarrow \tau_{\text{op}}$, so $t_r \Rightarrow \sigma(\tau_{\text{op}})$. Therefore, both S and S' satisfy constraints in Dyn-Movq-r-m and can execute the next store instruction.

Second, denote $e_a = i_d + e_b + e_i \times i_s$, $v_a = \sigma(e_a)$, and $v'_a = \sigma'(e_a) + \delta_0$. We construct the next states S_1 and S'_1 as follows:

$$pc_1 = \text{nextPc}(P,pc) \qquad pc'_1 = \text{nextPc}(C(P),pc')$$
$$M_1 = M[[v_a, v_a+8) \mapsto R[r_1]] \qquad M'_1 = M'[[v'_a, v'_a+8) \mapsto R'[r_1]]$$
$$\Phi_1 = (pc_1, (\sigma_f,\sigma_b)) \qquad \Phi'_1 = (pc'_1, (\sigma'_f,\sigma'_b))$$
$$S_1 = (R, M_1, \Phi_1) \qquad S'_1 = (R', M'_1, \Phi'_1).$$

According to Theorem 5, $P,\mathcal{P} \vdash_{\text{TAL}} S_1$.

Third, we prove that $P,\mathcal{P} \vdash S'_1 \prec S_1$. We begin with proving $P,\mathcal{P} \vdash S'_1 \prec_{\text{helper}} S_1$. According to Typing-Movq-r-m, there should exists s_o^{valid}, e_o, and τ_o such that

$$\Delta,\mathcal{R},\mathcal{M} \vdash \text{store}(i_d(r_b,r_i,i_s)^{s_{\text{op}},\tau_{\text{op}}}, 8, \mathcal{R}[r_1]) : (s_o^{\text{valid}}, (e_o,\tau_o)),$$

and for $\mathcal{M}_1 \coloneqq \mathcal{M}[s_{\text{op}} \mapsto (s_o^{\text{valid}}, (e_o,\tau_o))]$, we have $(\Delta,\mathcal{R},\mathcal{M}_1) = \text{getStateType}(\mathcal{P},pc_1)$. Then, according to Lemma 13, there should be $\text{getTransStrategy}(\mathcal{M}_1) = \text{getTransStrategy}(\mathcal{M}) = \omega$. Furthermore, both Typing-StoreOp-Spill and Typing-StoreOp-Non-Spill implies that $\tau_{\text{op}} = \tau_o$. This implies that $\delta_0 = \text{getOpShift}(\mathcal{M}_1,\omega,s_{\text{op}})$. According to Simulation-Relation-Helper, as all other requirements can be easily derived from $P,\mathcal{P} \vdash S' \prec S$, we just need to prove $\vdash (M'_1,\sigma') \prec_{\mathcal{M}_1} (M_1,\sigma)$.

Denote $ptr = \text{getPtr}(s_{\text{op}})$, then according to the store typing rules, $\Delta \vdash \text{isNonChangeExp}(e_a - ptr)$. Thus, there should be $\sigma'(e_a - ptr) = \sigma(e_a - ptr)$. Denote $\delta_p = \sigma'(ptr) - \sigma(ptr)$, then we have $v'_a = v_a + \delta_p + \delta_0$. Furthermore, the store typing rules also imply that $\Delta \vdash [e_a, e_a+8) \subseteq s_{\text{op}}$, so $[v_a, v_a+8) \subseteq \sigma(s_{\text{op}})$. Note that

$$\begin{aligned} &\text{getShiftedSlot}(\mathcal{M}_1,\omega,\sigma,\sigma',s_{\text{op}}) \\ &= \sigma(s_{\text{op}}) + \text{getOpShift}(\mathcal{M}_1,\omega,s_{\text{op}}) + \sigma'(s_{\text{op}}) - \sigma(s_{\text{op}}) \\ &= \sigma(s_{\text{op}}) + \delta_0 + \delta_p, \end{aligned}$$

so $[v'_a, v'_a+8) \subseteq \text{getShiftedSlot}(\mathcal{M}_1,\omega,\sigma,\sigma',s_{\text{op}})$.

For each $s \in \text{dom}(\mathcal{M}_1)$, denote

$$\mathcal{M}_1[s] = (s_1^{\text{valid}}, (e_1,\tau_1)) \qquad x = \text{getPtr}(s)$$
$$\delta_{\text{op1}} = \text{getOpShift}(\mathcal{M}_1,\omega,s) \qquad \delta_{\text{ptr}} = \sigma'(x) - \sigma(x).$$

According to Sim-Mem, we need to prove the following statements:

$$\omega(s) = \texttt{TransOp} \Rightarrow \sigma(s) \subseteq s_{\text{stackPub}} \wedge (\text{isLocalStack}(s) \vee \tau_1 \in \{0,1\})$$

$$\delta_{\text{ptr}} = \text{getPtrShift}(\mathcal{M}_1,\omega,\sigma,s)$$

$$s_1^{\text{valid}} = \emptyset \vee (M'[\sigma(s_1^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op1}}], \sigma') \prec_{e_1,\tau_1} (M[\sigma(s_1^{\text{valid}})], \sigma).$$

If $s \neq s_{\text{op}}$, then $\mathcal{M}_1[s] = \mathcal{M}[s]$ and $\Delta \vdash s \cap s_{\text{op}} = \emptyset$. According to Lemma 16, there should be

$$\text{getShiftedSlot}(\mathcal{M}_1,\omega,\sigma,\sigma',s) \cap \text{getShiftedSlot}(\mathcal{M}_1,\omega,\sigma,\sigma',s_{\text{op}}) = \emptyset$$

Note that $(\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op1}}) \subseteq \text{getShiftedSlot}(\mathcal{M}_1,\omega,\sigma,\sigma',s)$. Hence, we have

$$M_1[\sigma(s^{\text{valid}})] = M[\sigma(s^{\text{valid}})]$$
$$M'_1[\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op1}}] = M[\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op1}}].$$

We can then prove that the statement is true by using the above relation and the assumption $\vdash (M',\sigma') \prec_{\mathcal{M}} (M,\sigma)$ (derived from $P,\mathcal{P} \vdash S' \prec S$).

If $s = s_{\text{op}}$, we have

$$\mathcal{M}_1[s] = \mathcal{M}_1[s_{\text{op}}] = (s_o^{\text{valid}}, (e_o, \tau_{\text{op}})) = (s_1^{\text{valid}}, (e_1.\tau_1)).$$

The statement can be derived by discussing whether isSpill(s_{op}) and whether $[v_a, v_a + 8) \subseteq s_o^{\text{valid}}$. We omit details here since the reasoning is relatively straightforward.

If $f = main$, then $P, \mathcal{P} \vdash S_1' \prec S_1$ according to Simulation-Relation-Base.

We consider the case where $f \neq main$. The key idea is to prove that writing to $[v_a, v_a + 8)$ (or $[v_a', v_a' + 8)$) does not affect memory content not in getDom($\mathcal{M}, \sigma_f$) (or getShiftedDom($\mathcal{M}, \omega, \sigma_f, \sigma_f'$)).

There should exist pc_p, σ_{f_p}, σ_{b_p}, Φ_0, pc_p', σ_{f_p}', σ_{b_p}', and Φ_0' such that

$$\Phi = (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0 \qquad \Phi' = (pc_p', (\sigma_{f_p}', \sigma_{b_p}')); \Phi_0'.$$

Furthermore, $P, \mathcal{P} \vdash S' \prec S$ implies that there exists R_p, M_p, R_p', M_p' such that $P, \mathcal{P} \vdash (R_p', M_p', \Phi') \prec (R_p, M_p, \Phi)$. According to Lemma 14, we have

$$\begin{aligned}[v_a, v_a + 8) &\subseteq \sigma_{\text{op}}(s_{\text{op}}) \subseteq \text{getDom}(\mathcal{M}_1, \sigma_f)\\ [v_a', v_a' + 8) &\subseteq \text{getShiftedSlot}(\mathcal{M}_1, \omega, \sigma, \sigma', s_{\text{op}})\\ &\subseteq \text{getShiftedDom}(\mathcal{M}_1, \omega, \sigma_f, \sigma_f')\end{aligned}$$

From Lemma 7, we can also derive that $s_{\text{ret}} \cap \text{getDom}(\mathcal{M}_1, \sigma_f) = \emptyset$, and $s_{\text{ret}} \cap \text{getShiftedDom}(\mathcal{M}_1, \omega, \sigma_f, \sigma_f') = \emptyset$. Thus, there should be

$$M_1[s_{\text{ret}}] = M[s_{\text{ret}}] = (\text{nextPc}(P, pc_p), 0)$$
$$M_1'[s_{\text{ret}}] = M'[s_{\text{ret}}] = (\text{nextPc}(C(P), pc_p'), 0)$$
$$\forall x \notin \text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}.\ M_1[x] = M[x] = M_p[x]$$
$$\forall x \notin \text{getShiftedDom}(\mathcal{M}, \omega, \sigma_f, \sigma_f') \cup s_{\text{ret}}.\ M_1'[x] = M'[x] = M_p'[x].$$

Therefore, the statement also holds for the case when $f \neq main$.

Case jne $\ell^{\sigma_{\text{op}}}$ In this case, $C(inst) = inst$. First, according to Typing-Jne, flag ZF is untainted and we can denote its type as $\mathcal{R}[\text{ZF}] = (e = 0, 0)$, where $\Delta \vdash \text{isNonChangeExp}(e)$. Then, according to well-formedness of S and simulation relation between S' and S, there should be $R[\text{ZF}] = (v, 0)$, $R'[\text{ZF}] = (v', 0)$, and $\vdash ((v', 0), \sigma') \prec_{e=0,0} ((v, 0), \sigma)$. So both S and S' satisfy constraints in Dyn-Jne and can execute the next instruction. According to Sim-Val, it is straightforward to derive $v = v'$ by discussing on whether $e = \top$ and applying $\Delta \vdash \text{isNonChangeExp}(e)$. Thus, $inst$ and $C(inst)$ have the same branch direction (both taken or not taken).

Second, we construct the next states S_1 and S_1' as follows:

$$pc_n = \text{nextPc}(P, pc) \qquad pc_t = \text{getBlockPc}(P, \ell)$$
$$pc_n' = \text{nextPc}(C(P), pc') \qquad pc_t' = \text{getBlockPc}(C(P), \ell).$$

$$S_1 = \begin{cases}(R, M, (pc_n, (\sigma_f, \sigma_b); \Phi) & v = true\\ (R, M, (pc_t, (\sigma_f, \sigma \circ \sigma_{\text{op}}); \Phi) & v = false.\end{cases}$$

$$S_1' = \begin{cases}(R', M', (pc_n', (\sigma_f', \sigma_b'); \Phi') & v = true\\ (R', M', (pc_t', (\sigma_f', \sigma' \circ \sigma_{\text{op}}); \Phi') & v = false.\end{cases}$$

According to Theorem 5, there should be $P, \mathcal{P} \vdash_{\text{TAL}} S_1$.

Third, we prove that $P, \mathcal{P} \vdash S_1' \prec S_1$ under both cases (when the branch is taken and not taken). Denote $(\Delta, \mathcal{R}, \mathcal{M}) = \text{getStateType}(\mathcal{P}, pc)$. When $v = true$, i.e., the branch is not taken, all fields in S_1 (or S_1') are the same as S (or S') except for its current PC. The next state type is getStateType($\mathcal{P}, pc_n$) $= (\Delta \cup \{e = 0\}, \mathcal{R}, \mathcal{M})$ according to Typing-Jne, where the extra type context constraint is satisfied in both machines, i.e., $\sigma(e = 0) = \sigma'(e = 0) = true$. Furthermore, since $C(inst) = inst$, there should be $P, \mathcal{P} \vdash pc_n' \prec pc_n$. Therefore, we can easily derive $P, \mathcal{P} \vdash S_1' \prec S_1$ by unfolding $P, \mathcal{P} \vdash S' \prec S$.

When $v = false$, i.e., the branch is taken, according to Type-Jne, the next state type is $(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) \coloneqq \mathcal{P}(f)(\ell)$, where $\text{dom}(\mathcal{M}_1) = \text{dom}(\mathcal{M})$, and $(\Delta \cup \{e \neq 0\}, \mathcal{R}, \mathcal{M}) \sqsubseteq \sigma_{\text{op}}(\Delta_1, \mathcal{R}_1, \mathcal{M}_1)$. According to Lemma 13, getTransStrategy($\mathcal{M}_1$) = getTransStrategy($\mathcal{M}$) = ω.

We focus on proving $P, \mathcal{P} \vdash S_1' \prec_{\text{helper}} S_1$, as the other requirements of $P, \mathcal{P} \vdash S_1' \prec S_1$ can be easily derived by unfolding $P, \mathcal{P} \vdash S \prec S$ and applying what we have illustrated above. Denote

$$\sigma_1 = \sigma_f \cup (\sigma \circ \sigma_{\text{op}}) \qquad \sigma_1' = \sigma_f' \cup (\sigma' \circ \sigma_{\text{op}}).$$

According to Simulation-Relation-Helper, we just need to prove the following requirements:

- $P, \mathcal{P} \vdash pc_t' \prec pc_t$: this hold since both pc_t and pc_t' are PCs for block ℓ in P and $C(P)$.
- $P, \mathcal{P} \vdash (R, M, (\sigma_f, \sigma \circ \sigma_{\text{op}})) : (\Delta_1, \mathcal{R}_1, \mathcal{M}_1)$: this is implied by $P, \mathcal{P} \vdash_{\text{TAL}} S_1$.
- $\Delta_1 \vdash \sigma_f' \prec \sigma_f$: this is implied by $P, \mathcal{P} \vdash S' \prec S$.
- $\Delta_1 \vdash \sigma' \circ \sigma_{\text{op}} \prec \sigma \circ \sigma_{\text{op}}$: in this case, $\text{dom}(\sigma' \circ \sigma_{\text{op}}) = \text{dom}(\sigma' \circ \sigma_{\text{op}}) = \text{dom}(\sigma_{\text{op}})$. According to State-Subtype, we have $\Delta \vdash \sigma_{\text{op}}(\Delta_1)$. Hence, for all $x \in \text{dom}(\sigma_{\text{op}})$, if $\Delta_1 \vdash \text{isNonChangeOp}(x)$, then $\Delta \vdash \text{isNonChangeOp}(\sigma_{\text{op}}(x))$. Furthermore, according to the definition, it is easy to derive $\Delta \vdash \sigma' \prec \sigma$ from $\Delta \vdash \sigma_f' \prec \sigma_f$ and $\Delta \vdash \sigma_b' \prec \sigma_b$. Thus, $\sigma'(\sigma_{\text{op}}(x)) = \sigma(\sigma_{\text{op}}(x))$, and the original statement holds.
- $\vdash (R', \sigma_1') \prec_{\mathcal{R}_1} (R, \sigma_1)$: for all $r \in \mathcal{R}$, according to Sim-Reg, there should be $(R'[r], \sigma') \prec_{\mathcal{R}[r]} (R[r], \sigma)$. According to Reg-Subtype, we have $\vdash \mathcal{R}[r] \sqsubseteq \sigma_{\text{op}}(\mathcal{R}_1[r])$. Applying Lemma 18, we can get $\vdash (R'[r], \sigma' \circ \sigma_{\text{op}}) \prec_{\mathcal{R}_1[r]} (R[r], \sigma \circ \sigma_{\text{op}})$. Note that $\sigma_1 = \sigma \circ \sigma_{\text{op}}$ and $\sigma_1' = \sigma' \circ \sigma_{\text{op}}$ according to Lemma 9. Then, there should be $\vdash (R'[r], \sigma_1') \prec_{\mathcal{R}_1[r]} (R[r], \sigma_1)$. Thus, the statement is true.
- $\vdash (M', \sigma_1') \prec_{\mathcal{M}_1} (M, \sigma_1)$: for each $s_1 \in \text{dom}(\mathcal{M}_1)$, denote $\mathcal{M}_1[s_1] = (s_1^{\text{valid}}, (e_1, \tau_1))$ and $x_1 = \text{getPtr}(s_1)$. According to Typing-Jne, getTaintVar(dom(σ_{op})) = $\emptyset$, so $\sigma_{\text{op}}(\tau_1) = \tau_1$. According to State-Subtype and Mem-Slot-Subtype, there must exists $s \in \mathcal{M}_1$ and $\mathcal{M}[s] = (s^{\text{valid}}, (e, \tau))$ such that

$$\Delta \vdash \sigma_{\text{op}}(s_1) \subseteq s \quad \Delta \vdash \sigma_{\text{op}}(s_1^{\text{valid}}) \subseteq s^{\text{valid}}$$
$$\Delta \vdash \text{isSpill}(\sigma_{\text{op}}(s_1^{\text{valid}})) \Rightarrow \text{isSpill}(s^{\text{valid}})$$
$$\Delta \vdash e = \sigma_{\text{op}}(e_1) \vee (\text{isNonChangeExp}(e) \wedge \sigma_{\text{op}}(e_1) = \top) \vee \sigma_{\text{op}}(s_1^{\text{valid}}) = \emptyset$$
$$\Delta \vdash \tau = \tau_1 \vee (\text{isSpill}(s^{\text{valid}}) \wedge \sigma_{\text{op}}(s_1^{\text{valid}}) = \emptyset).$$

  Note that $\text{dom}(\mathcal{M}_1) = \text{dom}(\mathcal{M})$, so there should be $s_1 = s$ and $x_1 = x$. Reg-Mem-Type implies that getVars(s) = $\sigma_f(s)$, so $\sigma(s) = \sigma_f(s) = \sigma_1(s)$ and $\sigma(x) = \sigma_f(x) = \sigma_1(x)$. Denote

$\delta_{\text{op}} = \text{getOpShift}(\mathcal{M}_1, \omega, s)$ and $\delta_{\text{ptr}} = \sigma_1'(x) - \sigma_1(x)$. Hence, we just need to prove the following statements:

- $\omega(s) = \texttt{TransOp} \Rightarrow \sigma_1(s) \subseteq s_{\text{stackPub}} \wedge (\text{isLocalStack}(s) \vee \tau_1 \in \{0,1\})$: Suppose $\omega(s) = \texttt{TransOp}$. Then $\sigma(s) \subseteq s_{\text{stackPub}}$. Reg-Mem-Type implies that $\text{getVars}(s) = \sigma_f(s)$, so $\sigma(s) = \sigma_f(s) = \sigma_1(s)$. Thus, $\sigma_1(s) \subseteq s_{\text{stackPub}}$.
  Note that if $\neg\text{isLocalStack}(s)$, then $\tau \in \{0,1\}$ and $\neg\text{isSpill}(s)$, which is equivalent to $\neg\text{isSpill}(s^{\text{valid}})$. This together implies that $\tau = \tau_1$, so the statement is true.
- $\delta_{\text{ptr}} = \text{getPtrShift}(\mathcal{M}_1, \omega, \sigma_1, s)$: Note that $\delta_{\text{ptr}} = \sigma_1'(x) - \sigma_1(x) = \sigma'(x) - \sigma(x) = \text{getPtrShift}(\mathcal{M}, \omega, \sigma, s)$. Then, this can be proved by discussing whether $\omega(s) = \texttt{TransPtr}$. The key point is to note that when $\omega(s) = \texttt{TransPtr}$, there should be $\neg\text{isSpill}(s)$ and thereby $\tau = \tau_1$.
- $\sigma_1(s_1^{\text{valid}}) = \emptyset$ or $(M'[\sigma_1(s_1^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op}}], \sigma_1') \prec_{e_1,\tau_1} (M[\sigma_1(s_1^{\text{valid}})], \sigma_1)$: We just need to prove that the latter statement holds when $\sigma_1(s_1^{\text{valid}}) \neq \emptyset$. This implies that $\tau = \tau_1$, so we have

$$\delta_{\text{op}} = \text{getOpShift}(\mathcal{M}_1, \omega, s) = \text{getOpShift}(\mathcal{M}, \omega, s).$$

Then, from $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$, we can derive that

$$(M'[\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op}}], \sigma') \prec_{e,\tau} (M[\sigma(s^{\text{valid}})], \sigma).$$

We can then prove the original statement by discussing the format of e and applying Lemma 18, similar to the process of proving the simulation relation for registers.

Case callq $f_c^{\sigma_{\text{call}},\sigma_{\text{ret}}}$. Denote

$$pc_c = \text{getBlockPc}(P, f_c)$$
$$(\Delta_c, \mathcal{R}_c, \mathcal{M}_c) = \text{getStateType}(\mathcal{P}, pc_c)$$
$$\omega_c = \text{getTransStrategy}(\mathcal{M}_c).$$

According to the definition of C, there should be

$$C(inst) = \iota_1; \iota_2; \ldots; \iota_n; \textbf{callq}\ f_c^{\sigma'_{\text{call}},\sigma_{\text{ret}}}$$
$$((\iota_1; \iota_2; \ldots; \iota_n), \sigma'_{\text{call}}) = C_{\text{ptr}}((\Delta, \mathcal{R}, \mathcal{M}), \sigma_{\text{call}}(\Delta_c, \mathcal{R}_c, \mathcal{M}_c), \sigma_{\text{call}}),$$

and for $i = 1, \ldots, n$,

$$\iota_i = \textbf{addq}\ \delta, r_i \qquad r_i \neq r_{\texttt{rsp}}$$
$$\mathcal{R}_c[r_i] = (ptr_i, 0) \qquad \sigma'_{\text{call}}(ptr_i) = \sigma_{\text{call}}(ptr_i) + \delta$$

First, according to Typing-Callq, there exists a constant c such that $\mathcal{R}[r_{\texttt{rsp}}] = (sp + c, 0)$. Then, according Reg-Type, Value-Type, and well-formedness of S, there should be $R[r_{\texttt{rsp}}] = (v, 0)$ where $v = \sigma(sp) + c$. Thus, S satisfies constraints in Dyn-Callq and is allowed to execute *inst*. Furthermore, according to the simulation relation and the fact that sp is not affected by our transformation, i.e., $\text{isNonChangeExp}(sp)$, there should also be $R'[r_{\texttt{rsp}}] = (v, 0)$. Hence, S' can execute all instructions in $C(inst)$ without getting stuck.

Second, we construct the next states S_1 and S_1' as follows:

$$R_1 = R[r_{\texttt{rsp}} \mapsto (v - 8, 0)]$$
$$M_1 = M[[v - 8, v) \mapsto (\text{nextPc}(P, pc), 0)]$$
$$\Phi_1 = (pc_c, (\sigma \circ \sigma_{\text{call}}, [] \to [])); (pc, (\sigma_f, \sigma_b)); \Phi$$
$$pc_c' = \text{getBlockPc}(C(P), f_c) \quad pc'' = \text{getInstPc}(C(P), \textbf{callq}\ f_c^{\sigma'_{\text{call}},\sigma_{\text{ret}}})$$
$$R_1' = R'[r_{\texttt{rsp}} \mapsto (v - 8, 0), r_1 \mapsto (v_1 + \delta, 0), \ldots, r_n \mapsto (v_n + \delta, 0)]$$
$$M_1' = M'[[v - 8, v) \mapsto (\text{nextPc}(C(P), pc''), 0)]$$
$$\Phi_1' = (pc_c', (\sigma' \circ \sigma'_{\text{call}}, [] \to [])); (pc', (\sigma_f', \sigma_b')); \Phi'$$
$$S_1 = (R_1, M_1, \Phi_1) \qquad S_1' = (R_1', M_1', \Phi_1').$$

Third, we prove that $P, \mathcal{P} \vdash S_1' \prec S_1$ by proving all requirements in Simulation-Relation-Other are satisfied. We start with proving $P, \mathcal{P} \vdash S_1' \prec_{\text{helper}} S_1$. According to Simulation-Relation-Helper, as the other requirements are straightforward, we only prove the following statements:

- $\Delta_c \vdash \sigma_c' \prec \sigma_c$ where $\sigma_c' = \sigma' \circ \sigma'_{\text{call}}$ and $\sigma_c = \sigma \circ \sigma_{\text{call}}$: for all $x \in \text{dom}(\sigma_c)$ and $\Delta_c \vdash \text{isNonChangeExp}(x)$, there must be $x \neq ptr_i$ for $i = 1, \ldots, n$. Furthemore, according to State-Subtype and the subtype relation at call, there should be $\Delta \vdash \sigma_{\text{call}}(\Delta_c)$, which implies that $\Delta \vdash \text{isNonChangeExp}(\sigma_{\text{call}}(x))$. Thus, we have

$$\sigma_c'(x) = \sigma'(\sigma'_{\text{call}}(x)) = \sigma'(\sigma_{\text{call}}(x)) = \sigma(\sigma_{\text{call}}(x)) = \sigma_c(x),$$

  so the original statement holds.
- $\vdash (R_1', \sigma_c') \prec_{\mathcal{R}_c} (R_1, \sigma_c)$: this can be proved similarly to the case for **jne**, where the major difference is that we will apply Lemma 17 instead of Lemma 18.
- $\vdash (M_1', \sigma_c') \prec_{\mathcal{M}_c} (M_1, \sigma_c)$: this can be proved by applying Lemma 20.

Case retq. In this case, there should be $C(inst) = \textbf{retq}$. According to our assumption on the program, f must not be the top-level function, i.e., $f \neq main$. Then, we can denote

$$\Phi = (pc_p, (\sigma_{f_p}, \sigma_{b_p})); \Phi_0 \quad \Phi' = (pc_p', (\sigma_{f_p}', \sigma_{b_p}')); \Phi_0'.$$

According to Typing-Func, the state type of pc is $(\Delta, \mathcal{R}, \mathcal{M}) := \text{getStateType}(f)(f_{\text{ret}})$. It also implies that $\mathcal{R}[r_{\texttt{rsp}}] = (sp, 0)$. Then, according to Reg-Type, Value-Type, well-formedness of S, and simulation relation between S' and S, there should be $R[r_{\texttt{rsp}}] = R'[r_{\texttt{rsp}}] = (v, 0)$, where $v = \sigma(sp) = \sigma_f(sp)$. Furthermore, according to $P, \mathcal{P} \vdash S' \prec S$ and Simulation-Relation-Other, there should exist R_p, M_p, R_p', M_p' such that

$$s_{\text{ret}} = [\sigma_f(sp), \sigma(sp) + 8) = [v, v + 8)$$
$$M[s_{\text{ret}}] = (\text{nextPc}(P, pc_p), 0) \quad M'[s_{\text{ret}}] = (\text{nextPc}(C(P), pc_p'), 0)$$
$$\text{getInst}(P, pc_p) = \textbf{callq}\ f^{\sigma_{\text{call}},\sigma_{\text{ret}}} \quad \text{getInst}(C(P), pc_p') = \textbf{callq}\ f^{\sigma'_{\text{call}},\sigma_{\text{ret}}}$$
$$pc_{p0}' = \text{getCallPrefixPc}(C(P), pc_p')$$
$$\forall x \notin \text{getDom}(\mathcal{M}, \sigma_f) \cup s_{\text{ret}}.\ M[x] = M_p[x]$$
$$\forall x \notin \text{getShiftedDom}(\mathcal{M}, \omega, \sigma_f, \sigma_f') \cup s_{\text{ret}}.\ M'[x] = M_p'[x]$$
$$\sigma_f = (\sigma_{f_p} \cup \sigma_{b_p}) \circ \sigma_{\text{call}} \quad \sigma_f' = (\sigma_{f_p}' \cup \sigma_{b_p}') \circ \sigma'_{\text{call}}$$
$$P, \mathcal{P} \vdash (R_p', M_p', (pc_{p0}', (\sigma_{f_p}', \sigma_{b_p}')); \Phi_0') \prec (R_p, M_p, \Phi).$$

Thus, S and S' satisfy all constraints in Dyn-Retq and can execute **retq**.

Second, we construct the next states S_1 and S_1' as follows:

$$R_1 = R[r_{\mathtt{rsp}} \mapsto (v+8, 0)] \quad R_1' = R'[r_{\mathtt{rsp}} \mapsto (v+8, 0)]$$
$$\Phi_1 = (\mathrm{nextPc}(P, pc_p), (\sigma_{f_p}, (\sigma_b \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p}); \Phi_0$$
$$\Phi_1' = (\mathrm{nextPc}(C(P), pc_p'), (\sigma_{f_p}', (\sigma_b' \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p}'); \Phi_0$$
$$S_1 = (R_1, M, \Phi_1) \qquad S_1' = (R_1', M', \Phi_1').$$

Third, we prove that $P, \mathcal{P} \vdash S_1' \prec S_1$. As other requirements are relatively straightforward to prove, we only illustrate how to prove $P, \mathcal{P} \vdash S_1' \prec_{\mathrm{helper}} S_1$. Denote

$$(\Delta_0, \mathcal{R}_0, \mathcal{M}_0) = \mathrm{getStateType}(\mathcal{P}, pc_p)$$
$$(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) = (\sigma_{\mathrm{call}} \cup \sigma_{\mathrm{ret}})(\Delta, \mathcal{R}, \mathcal{M})$$
$$(\Delta_2, \mathcal{R}_2, \mathcal{M}_2) = (\Delta_0 \cup \Delta_1, \mathcal{R}_1[r_{\mathtt{rsp}} \mapsto \mathcal{R}_0[r_{\mathtt{rsp}}]], \mathrm{updateMem}(\mathcal{M}_0, \mathcal{M}_1))$$
$$\sigma_1 = \sigma_{f_p} \cup (\sigma_b \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p}$$
$$\sigma_1' = \sigma_{f_p}' \cup (\sigma_b' \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p}'.$$

According to Simulation-Relation-Helper, this holds as long as the following statements hold:

- $P, \mathcal{P} \vdash \mathrm{nextPc}(C(P), pc_p') \prec \mathrm{nextPc}(P, pc_p)$: $P, \mathcal{P} \vdash S' \prec S$ implies that $P, \mathcal{P} \vdash pc_{p0}' \prec pc_p$. According to the definition of pc_{p0}', the statement should hold since both PCs are referring to the next instruction after two matched calls.
- $(\Delta_2, \mathcal{R}_2, \mathcal{M}_2) = \mathrm{getStateType}(\mathcal{P}, \mathrm{nextPc}(P, pc_p))$ and $P, \mathcal{P} \vdash (R_1, M, \sigma_1) : (\Delta_2, \mathcal{R}_2, \mathcal{M}_2)$: this is implied from Theorem 5.
- $\Delta_2 \vdash \sigma_{f_p}' \prec \sigma_{f_p}$ and $\Delta_2 \vdash ((\sigma_b' \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p}') \prec ((\sigma_b \circ \sigma_{\mathrm{ret}}^{-1}) \cup \sigma_{b_p})$: The first one holds since $\mathrm{dom}(\sigma_{f_p}) \subseteq \mathrm{getVars}(\Delta_0)$ and $\Delta \vdash \sigma_{f_p} \prec \sigma_{f_p}'$ implied from $P, \mathcal{P} \vdash S' \prec S$. The second one can be proved by discussing about whether the variables belong to $\mathrm{dom}(\sigma_{b_p})$ or $\mathrm{dom}(\sigma_{\mathrm{ret}}^{-1})$.
- $\vdash (R_1', \sigma_1') \prec_{\mathcal{R}_2} (R_1, \sigma_1)$: We need to prove that for each $r \in \mathcal{R}_2$, $(R_1'[r], \sigma_1') \prec_{\mathcal{R}_2[r]} (R_1[r], \sigma_1)$. We consider the following three cases and briefly explain the idea to prove the statement:
  - $r = r_{\mathtt{rsp}}$: according to Typing-Callq, there should exist c such that $\mathcal{R}_2[r_{\mathtt{rsp}}] = \mathcal{R}_0[r_{\mathtt{rsp}}] = (sp + c, 0)$. Then, from Typing-Callq, there should be $\sigma_{\mathrm{call}}(sp) = sp + c - 8$. The relation can be proved by applying this relation and the assumption that isNonChangeExp(sp).
  - $r \neq r_{\mathtt{rsp}}$, and r is a callee-saved register: Typing-Func implies that in the callee function's type context, the dependent type of r is unchanged before and after the function call and is a non-top type variable. This implies that the value of r is also unchanged before and after executing f. We can then prove the statement by applying the above relation in the caller's type context.
  - $r \neq r_{\mathtt{rsp}}$, and r is not a callee-saved register: Typing-Func implies that isNonChangeExp($\mathcal{R}[r]$). We can then prove the statement by discussing whether the dependent type of r is $\top$ or not.
- $\vdash (M', \sigma_1') \prec_{\mathcal{M}_2} (M, \sigma_1)$: According to Lemma 13,

$$\omega_0 := \mathrm{getTransStrategy}(\mathcal{M}_2) = \mathrm{getTransStrategy}(\mathcal{M}),$$

so all assertions in getTransStrategy are also satisfied by $\mathcal{M}_2$.

For each $s \in \mathrm{dom}(\mathcal{M}_2)$, denote

$$\mathcal{M}_2[s] = (s_2^{\mathrm{valid}}, (e_2, \tau_2)) \qquad \mathcal{M}_0[s] = (s_0^{\mathrm{valid}}, (e_0, \tau_0))$$
$$\delta_s = \mathrm{getShift}(\mathcal{M}_2, \omega_0, \sigma_1, \sigma_1', s).$$

According to the assertions in updateMem, $\tau_2 = \tau_0$, so

$$\delta_s = \mathrm{getShift}(\mathcal{M}_2, \omega_0, \sigma_1, \sigma_1', s) = \mathrm{getShift}(\mathcal{M}_0, \omega_0, \sigma_1, \sigma_1', s).$$

According to Sim-Mem, as the other requirements of the simulation relation can be easily proved by unfolding $P, \mathcal{P} \vdash (R_p', M_p', (pc_{p0}', (\sigma_{f_p}', \sigma_{b_p}')); \Phi_0') \prec (R_p, M_p, \Phi)$, we focus on proving that if $\sigma_1(s_2^{\mathrm{valid}}) \neq \emptyset$, then

$$(M'[\sigma_1(s_2^{\mathrm{valid}}) + \delta_s], \sigma_1') \prec_{e_2, \tau_2} (M[\sigma_1(s_2^{\mathrm{valid}})], \sigma_1). \tag{5}$$

According to Algorithm 1, when executing updateMem to generate $\mathcal{M}_2$, we generate a map S that maps each slot in $\mathrm{dom}(\mathcal{M}_2)$ to slots in $\mathrm{dom}(\mathcal{M}_1)$ that belong to s. Consider the following cases:

- $s = \sigma_{\mathrm{call}}([sp, sp+8)) = [sp+c-8, sp+c)$ (where $\sigma_{\mathrm{call}}(sp) = sp+c-8$ and $\mathcal{R}_2[r_{\mathtt{rsp}}] = \mathcal{R}_0[r_{\mathtt{rsp}}] = (sp+c, \_)$): according to Typing-Callq, $\mathcal{M}_0[s] = (\emptyset, \_)$. According to Typing-Func, for all $s_1 \in \mathrm{dom}(\mathcal{M}_1)$, $s_1 \cap s = \emptyset$, so $S[s] = \emptyset$. These two facts imply that $s_2^{\mathrm{valid}} = \emptyset$, so $\sigma_1(s_2^{\mathrm{valid}}) = \emptyset$ and we do not need to worry about this case.
- $S[s] = \emptyset$ and $s \neq [sp + c - 8, sp + c)$: in this case, we have $s_2^{\mathrm{valid}} = s_0^{\mathrm{valid}}$ and $e_2 = e_0$. Since $\sigma_1(s_2^{\mathrm{valid}}) \neq \emptyset$, then $\sigma_1(s_0^{\mathrm{valid}}) \neq \emptyset$. According to the simulation relation between R_p, M_p and R_p', M_p', there should be

$$(M_p'[\sigma_1(s_0^{\mathrm{valid}}) + \delta_s], \sigma_1') \prec_{e_0, \tau_0} (M_p[\sigma_1(s_0^{\mathrm{valid}})], \sigma_1). \tag{6}$$

Recall that

$$\forall x \notin \mathrm{getDom}(\mathcal{M}, \sigma_f) \cup s_{\mathrm{ret}}.\ M[x] = M_p[x]$$
$$\forall x \notin \mathrm{getShiftedDom}(\mathcal{M}, \omega, \sigma_f, \sigma_f') \cup s_{\mathrm{ret}}.\ M'[x] = M_p'[x].$$

It is also straightforward to derive that

$$\sigma_1(s_2^{\mathrm{valid}}) \cap (\mathrm{getDom}(\mathcal{M}, \sigma_f) \cup s_{\mathrm{ret}}) = \emptyset$$
$$(\sigma_1(s_2^{\mathrm{valid}}) + \delta_s) \cap (\mathrm{getShiftedDom}(\mathcal{M}, \omega, \sigma_f, \sigma_f') \cup s_{\mathrm{ret}}) = \emptyset.$$

by applying Lemma 15, so $M'[\sigma_1(s_2^{\mathrm{valid}}) + \delta_s] = M_p'[\sigma_1(s_0^{\mathrm{valid}}) + \delta_s]$ and $M[\sigma_1(s_2^{\mathrm{valid}})] = M_p[\sigma_1(s_0^{\mathrm{valid}})]$. Thus, we can derive (5) from (6).

- cardinal($S[s]$) > 1: according to updateMem, $s_2^{\mathrm{valid}} = s_0^{\mathrm{valid}} = \emptyset$, so $\sigma_1(s_2^{\mathrm{valid}}) = \emptyset$ and we do not need to worry about this case.
- cardinal($S[s]$) $= 1$: denote $S[s] = \{s_1\}$ and $\mathcal{M}_1[s_1] = (s_1^{\mathrm{valid}}, (e_1, \tau_1))$. Then, $s_2^{\mathrm{valid}} = (s_0^{\mathrm{valid}} \backslash s_1) \cup s_1^{\mathrm{valid}}$. According to the definition of $\mathcal{M}_1$, there exists $s_c \in \mathcal{M}$ such that $s_c = (\sigma_{\mathrm{call}} \cup \sigma_{\mathrm{ret}})(s_1)$. Denote $\mathcal{M}[s_c] = (s_c^{\mathrm{valid}}, (e_c, \tau_c))$. Then, there should be $\mathcal{M}_1[s_1] = (\sigma_{\mathrm{call}} \cup \sigma_{\mathrm{ret}})(s_c^{\mathrm{valid}}, (e_c, \tau_c))$. Denote $(\Delta_c, \mathcal{R}_c, \mathcal{M}_c) = \mathrm{getStateType}(\mathcal{P}, \mathrm{getFuncPc}(P, f))$. According to assertions in updateMem (Algorithm 1), there

should be ¬isSpill(s_c). Then, by applying Lemma 12, 13, and 19, we can get

$$\delta_s = \text{getShift}(\mathcal{M}_2, \omega_0, \sigma_1, \sigma_1', s) = \text{getShift}(\mathcal{M}_0, \omega_0, \sigma_1, \sigma_1', s)$$
$$= \text{getShift}(\mathcal{M}_c, \omega, \sigma, \sigma', s_c) = \text{getShift}(\mathcal{M}, \omega, .\sigma, \sigma', s_c).$$

By unfolding the assumption $P, \mathcal{P} \vdash S' \prec S$, we can get

$$\sigma(s_c^{\text{valid}}) = \emptyset \vee (M'[\sigma(s_c^{\text{valid}}) + \delta_s], \sigma') \prec_{e_c, \tau_c} (M[\sigma(s_c^{\text{valid}})], \sigma). \quad (7)$$

According to Lemma 10, for all e, $\sigma(e) = \sigma_1((\sigma_{\text{call}} \cup \sigma_{\text{ret}})(e))$. Similarly, for all e such that $\sigma_{\text{call}}(e) = \sigma'_{\text{call}}(e)$, $\sigma'(e) = \sigma_1'((\sigma'_{\text{call}} \cup \sigma_{\text{ret}})(e)) = \sigma_1'((\sigma_{\text{call}} \cup \sigma_{\text{ret}})(e))$.
According to Typing-Func, we also have $s^{\text{valid}} = \emptyset \vee$ isNonChangeExp(e_c, τ_c). Then, according to the definition of C_{ptr} (Algorithm 5), $s^{\text{valid}} = \emptyset \vee \sigma_{\text{call}}(e_c) = \sigma'_{\text{call}}(e_c)$. Then, we can derive this from (7) by discussing whether e_c is $\top$ or not:

$$\sigma_1(s_1^{\text{valid}}) = \emptyset \vee (M'[\sigma_1(s_1^{\text{valid}}) + \delta_s], \sigma_1') \prec_{e_1, \tau_1} (M[\sigma_1(s_1^{\text{valid}})], \sigma_1). \quad (8)$$

We then consider the following cases.
If $s_2^{\text{valid}} = s_1^{\text{valid}}$, then $e_2 = e_1$. The statement can be proved by applying (8) and discussing whether $\sigma_1(s_1^{\text{valid}}) = \emptyset$ and whether $e_1 = \top$.
If $s_1^{\text{valid}} \subsetneq s_2^{\text{valid}}$, then $\Delta \vdash$ isNonChangeExp(e_0), $\Delta \vdash$ isNonChangeExp(e_1), and $e_2 = \top$. We only need to prove that for all $x \in \sigma_1(s_2^{\text{valid}})$, $M'[x + \delta_s] = M[x]$. If $x \in s_0^{\text{valid}} \backslash s_1$, we can prove this by applying (6) (which is derived from the simulation relation between R_p, M_p, and R'_p, M'_p) and $\Delta \vdash$ isNonChangeExp(e_0). If $x \in s_1^{\text{valid}}$, we can prove this by applying (8) and $\Delta \vdash$ isNonChangeExp(e_1).

□

Lemma 12. *For all $P, \mathcal{P}, pc_0, pc_1$, and for $i \in \{0, 1\}$, $(\Delta_i, \mathcal{R}_i, \mathcal{M}_i) = getStateType(\mathcal{P}, pc_i)$, if $getFunc(P, pc_0) = getFunc(P, pc_1)$, then for all $s \in \text{dom}(\mathcal{M}_0)$ such that ¬isSpill(s) and $\mathcal{M}_0[s] = (\_, (\_, \tau_0))$, $\mathcal{M}_1[s] = (\_, (\_, \tau_1))$, there must be $\tau_0 = \tau_1$.*

Proof. This lemma means that each non-spill memory slot has the same taint type at all instructions in the same function.

First, we prove that the statement holds when pc_0 and pc_1 refer to two consecutive instructions in the same basic block, i.e., $pc_1 = \text{nextPc}(P, pc_0)$. Denote $inst = \text{getInst}(P, pc_0)$, then according to our assumption, *inst* is not **jmp** or **retq**. Consider the following cases for *inst* (other cases can be proved similarly):

Case movq $r, i_d(r_d, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}$**.** For $s \neq s_{\text{op}}$, Typing-Movq-r-m implies that $\mathcal{M}_0[s] = \mathcal{M}_1[s]$, so $\tau_0 = \tau_1$.

For $s = s_{\text{op}}$, since ¬isSpill(s), then $\tau_0 = \tau_{\text{op}} = \tau_1$ according to Typing-Movq-r-m and Typing-StoreOp-Non-Spill.

Case movq $i_d(r_d, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}, r$**.** Typing-Movq-m-r implies that $\mathcal{M}_0 = \mathcal{M}_1$, so the statement is true.

Case jne $\ell^{\sigma_{\text{op}}}$**.** Since $pc_1 = \text{nextPc}(P, pc_0)$, pc_1 refers to the next instruction when the branch is not taken. Then, according to Typing-Jne, there should be $\mathcal{M}_0 = \mathcal{M}_1$, so the statement is true.

Case callq $f_c^{\sigma_{\text{call}}, \sigma_{\text{ret}}}$**.** Denote $(\Delta_c, \mathcal{R}_c, \mathcal{M}_c) = \mathcal{P}(f_c)(f_{\text{cret}})$. Then, $\mathcal{M}_1 = \text{updateMem}(\mathcal{M}_0, (\sigma_{\text{call}} \cup \sigma_{\text{ret}})(\mathcal{M}_c))$. According to the definition of updateMem (Algorithm 1), there should be $\tau_0 = \tau_1$.

Second, we prove that the statement holds when pc_0 refers to a branch instruction, and pc_1 is its branch target. We only illustrate the case where pc_0 refers to the branch instruction **jne** $\ell^{\sigma_{\text{op}}}$, since other cases can be proved similarly. Typing-Jne and State-Subtype imply that $\text{dom}(\mathcal{M}_0) = \text{dom}(\mathcal{M}_1)$ and $\mathcal{M}_0[s] \sqsubseteq \sigma_{\text{op}}(\mathcal{M}_1[s])$. Since ¬isSpill($s$), then Mem-Slot-Subtype implies that $\tau_0 = \sigma_{\text{op}}(\tau_1)$. According to Typing-Jne, $\text{getTaintVar}(\text{dom}(\sigma_{\text{op}})) = \emptyset$, so $\tau_0 = \sigma_{\text{op}}(\tau_1) = \tau_1$.

By combining the above two cases (pc_0 and pc_1 refer to consecutive instruction or branch/target instructions), we can further derive that the statement holds for all pc_0 and pc_1 that belong to the same function. □

Lemma 13. *For all $P, \mathcal{P}, pc_0, pc_1$, and for $i \in \{0, 1\}$, $(\Delta_i, \mathcal{R}_i, \mathcal{M}_i) = getStateType(\mathcal{P}, pc_i)$, if $getFunc(P, pc_0) = getFunc(P, pc_1)$, and $\mathcal{M}_0$ satisfies the assertion in getTransStrategy (Algorithm 3), then there should be $getTransStrategy(\mathcal{M}_0) = getTransStrategy(\mathcal{M}_1)$.*

Proof. This lemma means that memory state types of instructions in the same function have the same transformation strategy map. For each non-local-stack slot s, s is not a spill slot either, so $\mathcal{M}_0[s]$ and $\mathcal{M}_1[s]$ have the same taint type according to Lemma 12. Thus, $\mathcal{M}_1$ also satisfies the assertion in getTransStrategy (Algorithm 3).

Then, we can denote $\omega_i = \text{getTransStrategy}(\mathcal{M}_i)$, $i \in \{0, 1\}$. We aim to prove that $\omega_0 = \omega_1$.

Recall that in getTransStrategy (Algorithm 3), we first build a map from each base pointer to a set of taint types of slots referenced by the pointer. Denote the map for $\mathcal{M}_i$ as T_i, $i \in \{0, 1\}$. Note that if the base pointer of a slot is not *sp*, then the slot must not be a local stack slot. Hence, for all $ptr \in \text{dom}(T_1)$, $ptr \neq sp$, $T_0[ptr] = T_1[ptr]$.

For each $s \in \text{dom}(\mathcal{M}_0)$ with its base pointer denoted as $ptr = \text{getPtr}(s)$, we consider the following cases:

- $ptr \neq sp$ and $\text{cardinal}(T_0[ptr]) = 1$: $ptr \neq sp$ implies that $T_0[ptr] = T_1[ptr]$, so $\text{cardinal}(T_0[ptr]) = \text{cardinal}(T_1[ptr]) = 1$. Thus, $\omega_0(s) = \omega_1(s) = \texttt{TransPtr}$.
- $ptr \neq sp$ and $\text{cardinal}(T_0[ptr]) \neq 1$: $ptr \neq sp$ implies that $T_0[ptr] = T_1[ptr]$, so $\text{cardinal}(T_0[ptr]) = \text{cardinal}(T_1[ptr]) \neq 1$. Thus, $\omega_0(s) = \omega_1(s) = \texttt{TransOp}$.
- $ptr = sp$: in this case, there should be $\omega_0(s) = \omega_1(s) = \texttt{TransOp}$.

Therefore, $\omega_0 = \omega_1$. □

Lemma 14. *For all $\sigma, \sigma', \mathcal{M}$ and $\omega = getTransStrategy(\mathcal{M})$, if there exists M, M' such that $\vdash M : \sigma(\mathcal{M})$ and $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$, then for all $s \in \text{dom}(\mathcal{M})$,*

$$getShiftedSlot(\mathcal{M}, \omega, \sigma, \sigma', s) = \begin{cases} \sigma(s) \text{ or } \sigma(s) + \delta & \sigma(s) \subseteq s_{stackPub} \\ \sigma(s) & otherwise, \end{cases}$$

$$getPossibleSlot(\mathcal{M}, \omega, \sigma, \sigma', s) \subseteq \begin{cases} \sigma(s) \cup (\sigma(s) + \delta) & \sigma(s) \subseteq s_{stackPub} \\ \sigma(s) & otherwise, \end{cases}$$

and

$$getShiftedSlot(\mathcal{M}, \omega, \sigma, \sigma', s) \subseteq getPossibleSlot(\mathcal{M}, \omega, \sigma, \sigma', s).$$

Proof. Sim-Mem implies that

$$\begin{aligned} s' &:= \mathrm{getShiftedSlot}(\mathcal{M}, \omega, \sigma, \sigma', s) \\ &= \sigma(s) + \mathrm{getOpShift}(\mathcal{M}, \omega, s) + (\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))) \\ &= \sigma(s) + \mathrm{getOpShift}(\mathcal{M}, \omega, s) + \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s). \end{aligned}$$

We also denote

$$\begin{aligned} s'' &:= \mathrm{getPossibleSlot}(\mathcal{M}, \omega, \sigma, \sigma', s) \\ &= \begin{cases} \sigma(s) \cup (\sigma(s) + \delta) & \omega(s) = \texttt{TransOp} \\ \sigma(s) + (\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))) & \omega(s) = \texttt{TransPtr}. \end{cases} \end{aligned}$$

We consider the following three cases:

- $\sigma(s) \subseteq s_{\mathrm{stackPub}}$ and $\omega(s) = \texttt{TransOp}$:
$$\begin{aligned} s' &= \sigma(s) + \mathrm{getOpShift}(\mathcal{M}, \omega, s) + 0 \\ &= \sigma(s) \text{ or } \sigma(s) + \delta, \\ s'' &= \sigma(s) \cup (\sigma(s) + \delta). \end{aligned}$$
This implies that $s' \subseteq s''$.
- $\sigma(s) \subseteq s_{\mathrm{stackPub}}$ and $\omega(s) = \texttt{TransPtr}$:
$$\begin{aligned} s' &= \sigma(s) + 0 + (\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))) \\ &= \sigma(s) + \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) \\ &= \sigma(s) \text{ or } \sigma(s) + \delta \\ s'' &= \sigma(s) + (\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))) = s' \\ &= \sigma(s) + \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) \\ &\subseteq \sigma(s) \cup (\sigma(s) + \delta). \end{aligned}$$
- $\sigma(s) \nsubseteq s_{\mathrm{stackPub}}$: this implies that
$$\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s)) = \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) = 0.$$
Sim-Mem requires $\omega(s) = \texttt{TransOp} \Rightarrow \sigma(s) \subseteq s_{\mathrm{stackPub}}$, so in this case $\omega(s) \neq \texttt{TransOp}$, i.e., $\omega(s) = \texttt{TransPtr}$. Therefore,
$$\begin{aligned} s' &= \sigma(s) + 0 + 0 = \sigma(s) \\ s'' &= \sigma(s) + 0 = \sigma(s) = s' \subseteq \sigma(s) \cup (\sigma(s) + \delta). \end{aligned}$$

□

Lemma 15. *For all σ, σ', $\mathcal{M}$ and $\omega = \mathit{getTransStrategy}(\mathcal{M})$, if there exists M, M' such that $\vdash M : \sigma(\mathcal{M})$ and $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$, then for all $s_1, s_2 \in \mathrm{dom}(\mathcal{M})$, $\sigma(s_1) \cap \sigma(s_2) = \emptyset$, denoting*

$$\begin{aligned} s_1' &= \mathit{getPossibleSlot}(\mathcal{M}, \omega, \sigma, \sigma', s_1) \\ s_2' &= \mathit{getPossibleSlot}(\mathcal{M}, \omega, \sigma, \sigma', s_2), \end{aligned}$$

there should be $s_1' \cap s_2' = \emptyset$.

Proof. Mem-Type implies that for $i \in \{1, 2\}$, $\sigma(s_i) \subseteq s_{\mathrm{stackPub}} \vee \sigma(s_i) \subseteq s_{\mathrm{otherPub}} \cup s_{\mathrm{otherSec}}$. We then consider the following four cases about s_1 and s_2:

- $\sigma(s_1) \subseteq s_{\mathrm{stackPub}}$ and $\sigma(s_2) \subseteq s_{\mathrm{stackPub}}$: this implies that $(\sigma(s_1) + \delta) \subseteq s_{\mathrm{stackSec}}$ and $(\sigma(s_2) + \delta) \subseteq s_{\mathrm{stackSec}}$. Since $s_{\mathrm{stackPub}} \cap s_{\mathrm{stackSec}} = \emptyset$ and $\sigma(s_1) \cap \sigma(s_2) = \emptyset$, then
$$\begin{aligned} s_1' \cap s_2' &\subseteq (\sigma(s_1) \cup (\sigma(s_1) + \delta)) \cap (\sigma(s_2) \cup (\sigma(s_2) + \delta)) \\ &= (\sigma(s_1) \cap \sigma(s_2)) \cup (\sigma(s_1) \cap (\sigma(s_2) + \delta)) \\ &\quad \cup ((\sigma(s_1) + \delta) \cap \sigma(s_2)) \cup ((\sigma(s_1) + \delta) \cap (\sigma(s_2) + \delta)) \\ &= \emptyset \cup \emptyset \cup \emptyset \cup \emptyset = \emptyset. \end{aligned}$$
- $\sigma(s_1) \subseteq s_{\mathrm{stackPub}}$ and $\sigma(s_2) \nsubseteq s_{\mathrm{stackPub}}$:
$$\begin{aligned} s_1' \cap s_2' &\subseteq (\sigma(s_1) \cup (\sigma(s_1) + \delta)) \cap \sigma(s_2) \\ &\subseteq (s_{\mathrm{stackPub}} \cup s_{\mathrm{stackSec}}) \cap (s_{\mathrm{otherPub}} \cup s_{\mathrm{otherSec}}) \\ &= \emptyset. \end{aligned}$$
- $\sigma(s_1) \nsubseteq s_{\mathrm{stackPub}}$ and $\sigma(s_2) \subseteq s_{\mathrm{stackPub}}$: the proof is similar to the last case.
- $\sigma(s_1) \nsubseteq s_{\mathrm{stackPub}}$ and $\sigma(s_2) \nsubseteq s_{\mathrm{stackPub}}$:
$$s_1' \cap s_2' \subseteq \sigma(s_1) \cap \sigma(s_2) = \emptyset.$$

□

Lemma 16. *For all σ, σ', $\mathcal{M}$ and $\omega = \mathit{getTransStrategy}(\mathcal{M})$, if there exists M, M' such that $\vdash M : \sigma(\mathcal{M})$ and $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$, then for all $s_1, s_2 \in \mathrm{dom}(\mathcal{M})$, $\sigma(s_1) \cap \sigma(s_2) = \emptyset$, denoting*

$$\begin{aligned} s_1' &= \mathit{getShiftedSlot}(\mathcal{M}, \omega, \sigma, \sigma', s_1) \\ s_2' &= \mathit{getShiftedSlot}(\mathcal{M}, \omega, \sigma, \sigma', s_2), \end{aligned}$$

there should be

$$s_1' \cap s_2' = s_1' \cap \sigma(s_2) = \sigma(s_1) \cap s_2' = \emptyset.$$

Proof. Denote $s_i'' = \mathrm{getPossibleSlot}(\mathcal{M}, \omega, \sigma, \sigma', s_i)$, $i \in \{1, 2\}$. According to Lemma 14, there should be $\sigma(s_i) \subseteq s_i''$ and $s_i' \subseteq s_i''$. Thus, the statement can be proved by applying Lemma 15. □

Lemma 17. *For all $e, \tau, v, t, v', t', \sigma, \sigma', \Delta$, if*

$$\vdash ((v', t'), \sigma') \prec_{e,\tau} ((v, t), \sigma) \quad \Delta \vdash \sigma' \prec \sigma,$$

then for all σ_m, σ_m', e_1, τ_1, Δ_1 where

$$\begin{aligned} &\Delta \vdash e = \sigma_m(e_1) \vee (\mathit{isNonChangeExp}(e) \wedge \sigma_m(e_1) = \top) \\ &\Delta \vdash \sigma_m(\Delta_1) \qquad \forall x \in \mathrm{dom}(\sigma_m).\ \sigma_m(x) \neq \top \end{aligned}$$

the following statements hold:

- *if $\sigma_m'(e_1) = \sigma_m(e_1) = \top$, then $\vdash ((v', t'), \sigma' \circ \sigma_m') \prec_{e_1,\tau_1} ((v, t), \sigma \circ \sigma_m)$;*
- *if $\sigma_m'(e_1) = \sigma_m(e_1) + c \neq \top$ for some constant number c, then $\vdash ((v' + c, t'), \sigma' \circ \sigma_m') \prec_{e_1,\tau_1} ((v, t), \sigma \circ \sigma_m)$.*

Proof. Denote $\sigma_1 = \sigma \circ \sigma_m$, $\sigma_1' = \sigma' \circ \sigma_m'$. Consider the following cases:

- $\sigma_m'(e_1) = \sigma_m(e_1) = \top$: according to Sim-Val and our assumption, there should be $t = t'$, so we just need to prove that
$$(e_1 = \top \wedge v' = v) \vee (e_1 \neq \top \wedge v' = \sigma_1'(e_1) \wedge v = \sigma_1(e_1)).$$
Since for all $x \in \mathrm{dom}(\sigma_m)$, $\sigma_m(x) \neq \top$, then $\sigma_m(e_1) = \top$ implies that $e_1 = \top$. Consider the following cases:
  - $e = \top$: according to Sim-Val, $\vdash ((v', t'), \sigma') \prec_{e,\tau} ((v, t), \sigma)$ implies that $v = v'$, so the statement is true.
  - $e \neq \top$: in this case, $\sigma_m(e_1) = \top \neq e$. Then, according to our assumption, there must be isNonChangeExp(e), so $v' = \sigma'(e) = \sigma(e) = v$ and the statement is true.

- $\sigma'_m(e_1) = \sigma_m(e_1) + c \neq \top$: according to Sim-Val and our assumption, there should be $t = t'$, so we just need to prove that

$$(e_1 = \top \wedge v' + c = v) \vee (e_1 \neq \top \wedge v' + c = \sigma'_1(e_1) \wedge v = \sigma_1(e_1)).$$

According to our assumption, there must be $\sigma_m(e_1) = e$ and $\sigma'_m(e_1) = \sigma_m(e_1) + c = e + c$. Then,

$$\sigma_1(e_1) = \sigma(e) = v \qquad \sigma'_1(e_1) = \sigma'(e) + c = v' + c.$$

Thus, the statement is true.

□

Lemma 18. *For all $e, \tau, v, t, v', t', \sigma, \sigma', \Delta$, if*

$$\vdash ((v', t'), \sigma') \prec_{e,\tau} ((v, t), \sigma) \quad \Delta \vdash \sigma' \prec \sigma,$$

then for all $\sigma_m, e_1, \tau_1, \Delta_1$ where

$$\Delta \vdash e = \sigma_m(e_1) \vee (\mathit{isNonChangeExp}(e) \wedge \sigma_m(e_1) = \top)$$
$$\Delta \vdash \sigma_m(\Delta_1) \qquad \forall x \in \mathrm{dom}(\sigma_m).\ \sigma_m(x) \neq \top,$$

then there should be

$$\vdash ((v', t'), \sigma' \circ \sigma_m) \prec_{e_1,\tau_1} ((v, t), \sigma \circ \sigma_m).$$

Proof. This can be proved by applying Lemma 17 with $\sigma'_m = \sigma_m$ and $c = 0$. □

Lemma 19. *For all $(\Delta, \mathcal{R}, \mathcal{M}), (\Delta_1, \mathcal{R}_1, \mathcal{M}_1), \sigma, \sigma', \sigma_1, \sigma'_1, \sigma_m, \sigma'_m$, if*

$$\sigma(\Delta) = \mathit{true} \quad \Delta \vdash \sigma' \prec \sigma \quad \sigma_1 = \sigma \circ \sigma_m \quad \sigma'_1 = \sigma' \circ \sigma'_m$$
$$\Delta \vdash \sigma_m(\Delta_1) \quad \forall x \in \mathrm{dom}(\sigma_m).\ \sigma_m(x) \neq \top \quad \sigma_m(\mathit{sp}) = \mathit{sp} + c$$
$$\forall x, \sigma_m(x) = e, \mathit{isPtr}(x).\ \Delta \vdash \mathit{isNonChangeExp}(e - \mathit{getPtr}(e))$$
$$(\_, \sigma'_m) = C_{ptr}((\Delta, \mathcal{R}, \mathcal{M}), \sigma_m(\Delta_1, \mathcal{R}_1, \mathcal{M}_1), \sigma_m)$$
$$\omega = \mathit{getTransStrategy}(\mathcal{M}) \quad \omega_1 = \mathit{getTransStrategy}(\mathcal{M}_1),$$

then for all s, s_1 where

$$\Delta \vdash \sigma_m(s_1) \subseteq s \wedge \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1) \quad \neg \mathit{isSpill}(s),$$

then there should be

$$\mathit{getShift}(\mathcal{M}_1, \omega_1, \sigma_1, \sigma'_1, s_1) = \mathit{getShift}(\mathcal{M}, \omega, \sigma, \sigma', s).$$

Proof. According to the definition of getShift, we just need to prove that

$$\sigma'_1(\mathrm{getPtr}(s_1)) - \sigma_1(\mathrm{getPtr}(s_1)) + \mathrm{getOpShift}(\mathcal{M}_1, \omega_1, s_1)$$
$$= \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s)) + \mathrm{getOpShift}(\mathcal{M}, \omega, s).$$

Denote $\mathrm{getPtr}(s_1) = x_1$, $\sigma_m(x_1) = e_x$. According to our assumption, $\Delta \vdash \mathrm{isNonChangeExp}(e_x - \mathrm{getPtr}(e_x))$, so $\sigma'(e_x - \mathrm{getPtr}(e_x)) = \sigma(e_x - \mathrm{getPtr}(e_x))$.

Furthermore, according to Mem-Slot-Subtype, $\Delta \vdash \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1)$ and $\neg\mathrm{isSpill}(s)$ imply that $\sigma_1(\tau_1) = \sigma(\tau)$.

Following the definition of C_{ptr} (Algorithm 5), we just need to consider the following cases:

- $\omega_1(s_1) = \omega(s)$: according to the definition of C_{ptr} (Algorithm 5), $\sigma'_m(x_1) = \sigma_m(x_1) = e_x$, so

$$\begin{aligned}\sigma'_1(x_1) - \sigma_1(x_1) &= \sigma'(\sigma'_m(x_1)) - \sigma(\sigma_m(x_1)) \\ &= \sigma'(e_x) - \sigma(e_x) \\ &= \sigma'(\mathrm{getPtr}(e_x)) - \sigma(\mathrm{getPtr}(e_x)) \\ &= \sigma'(\mathrm{getPtr}(\sigma_m(s_1))) - \sigma(\mathrm{getPtr}(\sigma_m(s_1))) \\ &= \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s)).\end{aligned}$$

According to the definition of getOpShift, $\sigma_1(\tau_1) = \sigma(\tau)$ and $\omega_1(s_1) = \omega(s)$ implies that

$$\mathrm{getOpShift}(\mathcal{M}_1, \omega_1, s_1) = \mathrm{getOpShift}(\mathcal{M}, \omega, s).$$

Therefore, the statement holds.

- $\omega_1(s_1) = \texttt{TransPtr}$, $\omega(s) = \texttt{TransOp}$: If $\sigma_m(\tau_1) \neq 0$, then $\tau \neq 0$. According to the definition of C_{ptr}, $\sigma'_m(x_1) = \sigma_m(x_1) + \delta = e_x + \delta$. Then,

$$\begin{aligned}\sigma'_1(x_1) - \sigma_1(x_1) &= \sigma'(\sigma'_m(x_1)) - \sigma(\sigma_m(x_1)) \\ &= \sigma'(e_x + \delta) - \sigma(e_x) \\ &= \delta + \sigma'(\mathrm{getPtr}(e_x)) - \sigma(\mathrm{getPtr}(e_x)) \\ &= \delta + \sigma'(\mathrm{getPtr}(\sigma_m(s_1))) - \sigma(\mathrm{getPtr}(\sigma_m(s_1))) \\ &= \delta + \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s)).\end{aligned}$$

Furthermore, in this case there should be

$$\mathrm{getOpShift}(\mathcal{M}_1, \omega_1, s_1) = 0 \quad \mathrm{getOpShift}(\mathcal{M}, \omega, s) = \delta.$$

Therefore, the statement holds.

If $\sigma_m(\tau_1) = 0$, then $\tau = 0$. According to the definition of C_{ptr}, $\sigma'_m(x_1) = \sigma(x_1) = e_x$. Similar to the first case, we have $\sigma'_1(x_1) - \sigma_1(x_1) = \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))$. Furthermore, there should also be

$$\mathrm{getOpShift}(\mathcal{M}_1, \omega_1, s_1) = \mathrm{getOpShift}(\mathcal{M}, \omega, s) = 0.$$

Therefore, the statement holds.

□

Lemma 20. *For all $\mathcal{M}, M, M', \sigma, \sigma', \Delta, \mathcal{R}$, if*

$$\sigma(\Delta) = \{\mathit{true}\} \quad \vdash M : \sigma(\mathcal{M}) \quad \vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma) \quad \Delta \vdash \sigma' \prec \sigma,$$

then for all $\mathcal{M}_1, \sigma_m, \sigma'_m, \Delta_1, \mathcal{R}_1$, where $\mathcal{M}_1$ satisfies the assertion in getTransStrategy (Algorithm 3) and

$$\forall s_1 \in \mathrm{dom}(\mathcal{M}_1).\ \exists s.\ \Delta \vdash \sigma_m(s_1) \subseteq s \wedge \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1[s_1])$$
$$\Delta \vdash \sigma_m(\Delta_1) \quad \forall x \in \mathrm{dom}(\sigma_m).\ \sigma_m(x) \neq \top \quad \sigma_m(\mathit{sp}) = \mathit{sp} + c,$$
$$\forall x, \sigma_m(x) = e, \mathit{isPtr}(x).\ \Delta \vdash \mathit{isNonChangeExp}(e - \mathit{getPtr}(e))$$
$$(\_, \sigma'_m) = C_{ptr}((\Delta, \mathcal{R}, \mathcal{M}), \sigma_m(\Delta_1, \mathcal{R}_1, \mathcal{M}_1), \sigma_m),$$

then there should be

$$\vdash (M', \sigma' \circ \sigma'_m) \prec_{\mathcal{M}_1} (M, \sigma \circ \sigma_m).$$

Proof. Denote

$$\omega = \mathrm{getTransStrategy}(\mathcal{M}) \qquad \omega_1 = \mathrm{getTransStrategy}(\mathcal{M}_1)$$
$$\sigma_1 = \sigma \circ \sigma_m \qquad \sigma'_1 = \sigma' \circ \sigma'_m.$$

To prove the lemma, we need to prove that for each $s_1 \in \mathrm{dom}(\mathcal{M}_1)$, where

$$\mathcal{M}_1[s_1] = (s_1^{\mathrm{valid}}, (e_1, \tau_1)) \qquad x_1 = \mathrm{getPtr}(s_1)$$
$$\delta_{\mathrm{op1}} = \mathrm{getOpShift}(\mathcal{M}_1, \omega_1, s_1) \qquad \delta_{\mathrm{ptr1}} = \sigma_1'(x_1) - \sigma_1(x_1),$$

the following statements hold:

- $\omega_1(s_1) = \texttt{TransOp} \Rightarrow \sigma_1(s_1) \subseteq s_{\mathrm{stackPub}} \wedge (\mathrm{isLocalStack}(s_1) \vee \tau_1 \in \{0, 1\})$;
- $\delta_{\mathrm{ptr1}} = \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1)$;
- $\sigma_1(s_1^{\mathrm{valid}}) = \emptyset \vee ((M'[\sigma_1(s_1^{\mathrm{valid}}) + \delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}}], \sigma_1') \prec_{e_1,\tau_1} (M[\sigma_1(s_1^{\mathrm{valid}})], \sigma_1))$.

According to our assumption, there must exists $s \in \mathrm{dom}(\mathcal{M})$ such that $\Delta \vdash \sigma_m(s_1) \subseteq s \wedge \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1[s_1])$. Denote $\mathcal{M}[s] = (s^{\mathrm{valid}}, (e, \tau))$. We prove the three statements as follows:

- $\omega_1(s_1) = \texttt{TransOp} \Rightarrow \sigma_1(s_1) \subseteq s_{\mathrm{stackPub}} \wedge (\mathrm{isLocalStack}(s_1) \vee \tau_1 \in \{0, 1\})$: according to our assumption, $\mathcal{M}_1$ satisfies the assertion in Algorithm 3, so when $\omega_1(s_1) = \texttt{TransOp}$, $\mathrm{isLocalStack}(s_1) \vee \tau_1 \in \{0, 1\}$.
  We just need to prove that when $\omega_1(s_1) = \texttt{TransOp}$, $\sigma_1(s_1) \subseteq s_{\mathrm{stackPub}}$. Since $\Delta \vdash \sigma_m(s_1) \subseteq s$, then $\sigma_1(s_1) = (\sigma \circ \sigma_m)(s_1) \subseteq \sigma(s)$.
  Furthermore, according to the assertion in C_{ptr} (Algorithm 5), when $\omega_1(s_1) = \texttt{TransOp}$, there should be $\omega(s) = \texttt{TransOp}$. Then, $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$ and Sim-Mem implies that $\sigma(s) \subseteq s_{\mathrm{stackPub}}$. Therefore, $\sigma_1(s_1) \subseteq \sigma(s) \subseteq s_{\mathrm{stackPub}}$.
- $\delta_{\mathrm{ptr1}} = \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1)$: denote $\sigma_m(x_1) = e_x$. According to our assumption, $\Delta \vdash \mathrm{isNonChangeExp}(e_x - \mathrm{getPtr}(e_x))$, so $\sigma'(e_x - \mathrm{getPtr}(e_x)) = \sigma(e_x - \mathrm{getPtr}(e_x))$. According to Sim-Mem, $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$ implies that $\sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s)) = \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s)$. Consider the following cases:
  - $\omega_1(s_1) = \omega(s) = \texttt{TransOp}$: according to the definition of C_{ptr} (Algorithm 5), $\sigma_m'(x_1) = \sigma_m(x_1) = e_x$. Then,
    $$\begin{aligned}\delta_{\mathrm{ptr1}} &= \sigma_1'(x_1) - \sigma_1(x_1)\\ &= \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))\\ &= \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) = 0\\ &= \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1),\end{aligned}$$
    where the second step can be derived similarly to the proof of Lemma 19.
  - $\omega_1(s_1) = \omega(s) = \texttt{TransPtr}$: according to the definition of C_{ptr} (Algorithm 5), $\sigma_m'(x_1) = \sigma_m(x_1) = e_x$. According to Algorithm 3, s_1 and s are not referenced by the stack pointer, so they are not spill slots. Then, $\Delta \vdash \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1[s_1])$ implies that $\Delta \vdash \tau = \sigma_m(\tau_1)$, so $\sigma_1(\tau_1) = \sigma(\tau)$. Then, we can derive
    $$\begin{aligned}\delta_{\mathrm{ptr1}} &= \sigma_1'(x_1) - \sigma_1(x_1)\\ &= \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))\\ &= \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s)\\ &= \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1),\end{aligned}$$
    where the second step can be derived similarly to the proof of Lemma 19, and the last step holds because $\sigma_1(\tau_1) = \sigma(\tau)$ and $\sigma_1(s_1) \subseteq \sigma(s)$.
  - $\omega_1(s_1) = \texttt{TransPtr}$, $\omega(s) = \texttt{TransOp}$: if $\sigma_m(\tau_1) \neq 0$, according to the definition of C_{ptr} (Algorithm 5), $\sigma_m'(x_1) = \sigma_m(x_1) + \delta = e_x + \delta$.
    $$\begin{aligned}\delta_{\mathrm{ptr1}} &= \sigma_1'(x_1) - \sigma_1(x_1)\\ &= \delta + \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))\\ &= \delta + \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) = \delta\\ &= \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1),\end{aligned}$$
    where the second step can be derived similarly to the proof of Lemma 19.
    If $\sigma_m(\tau_1) = 0$, then $\sigma_m'(x_1) = \sigma_m(x_1) = e_x$ and $\sigma_1(\tau_1) = \sigma(\sigma_m(\tau_1)) = 0$. So
    $$\begin{aligned}\delta_{\mathrm{ptr1}} &= \sigma_1'(x_1) - \sigma_1(x_1)\\ &= \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))\\ &= \mathrm{getPtrShift}(\mathcal{M}, \omega, \sigma, s) = 0\\ &= \mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1),\end{aligned}$$
    where the second step can be derived similarly to the proof of Lemma 19, and the last step holds since $\sigma_1(\tau_1) = 0$ implies that $\mathrm{getPtrShift}(\mathcal{M}_1, \omega_1, \sigma_1, s_1) = 0$ or δ.
  Therefore, the statement is true.
- $\sigma_1(s_1^{\mathrm{valid}}) = \emptyset \vee ((M'[\sigma_1(s_1^{\mathrm{valid}}) + \delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}}], \sigma_1') \prec_{e_1,\tau_1} (M[\sigma_1(s_1^{\mathrm{valid}})], \sigma_1))$: if $\sigma_1(s_1^{\mathrm{valid}}) = \emptyset$, the statement holds.
  If $\mathrm{isSpill}(s)$, then according to the assertions in C_{ptr} (Algorithm 5), $s_1^{\mathrm{valid}} = \emptyset$, so $\sigma_1(s_1^{\mathrm{valid}}) = \emptyset$ and the statement holds.
  If $\neg\mathrm{isSpill}(s)$ and $\sigma_1(s_1^{\mathrm{valid}}) \neq \emptyset$, we need to prove that

$$(M'[\sigma_1(s_1^{\mathrm{valid}}) + \delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}}], \sigma_1') \prec_{e_1,\tau_1} (M[\sigma_1(s_1^{\mathrm{valid}})], \sigma_1).$$

  Denote $\delta_{\mathrm{op}} = \mathrm{getOpShift}(\mathcal{M}, \omega, s)$, $\delta_{\mathrm{ptr}} = \sigma'(\mathrm{getPtr}(s)) - \sigma(\mathrm{getPtr}(s))$. In this case, we can apply Lemma 19 and get that $\delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}} = \delta_{\mathrm{ptr}} + \delta_{\mathrm{op}}$. Since $\sigma(\sigma_m(s_1^{\mathrm{valid}})) = \sigma_1(s_1^{\mathrm{valid}}) \neq \emptyset$, then $\sigma_m(s_1^{\mathrm{valid}}) \neq \emptyset$. Then, $\Delta \vdash \mathcal{M}[s] \sqsubseteq \sigma_m(\mathcal{M}_1[s_1])$ and Mem-Slot-Subtype imply that

$$\Delta \vdash \sigma_m(s_1^{\mathrm{valid}}) \subseteq s^{\mathrm{valid}}$$
$$\Delta \vdash e = \sigma_m(e_1) \vee (\mathrm{isNonChangeExp}(e) \wedge \sigma_m(e_1) = \top).$$

  Then, we can get $\sigma_1(s_1^{\mathrm{valid}}) = \sigma(\sigma_m(s_1^{\mathrm{valid}})) \subseteq \sigma(s^{\mathrm{valid}})$. Since $\sigma_1(s_1^{\mathrm{valid}}) \neq \emptyset$, then $\sigma(s^{\mathrm{valid}}) \neq \emptyset$. According to Sim-Mem, $\vdash (M', \sigma') \prec_{\mathcal{M}} (M, \sigma)$ implies that

$$(M'[\sigma(s^{\mathrm{valid}}) + \delta_{\mathrm{ptr}} + \delta_{\mathrm{op}}], \sigma') \prec_{e,\tau} (M[\sigma(s^{\mathrm{valid}})], \sigma). \tag{9}$$

  Consider the following cases:
  - $\Delta \vdash \sigma_m(s_1^{\mathrm{valid}}) = s^{\mathrm{valid}}$: in this case, $\sigma_1(s_1^{\mathrm{valid}}) = \sigma(s^{\mathrm{valid}})$, so $M[\sigma_1(s_1^{\mathrm{valid}})] = M[\sigma(s^{\mathrm{valid}})]$. Furthermore, since $\delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}} = \delta_{\mathrm{ptr}} + \delta_{\mathrm{op}}$, then $M'[\sigma_1(s_1^{\mathrm{valid}}) + \delta_{\mathrm{ptr1}} + \delta_{\mathrm{op1}}] = M'[\sigma(s^{\mathrm{valid}}) + \delta_{\mathrm{ptr}} + \delta_{\mathrm{op}}]$. According to the assertions in C_{ptr} (Algorithm 5), there should be $\sigma_m(e_1) = \sigma_m'(e_1)$. Thus, the statement can be proved by applying Lemma 17.
  - $\Delta \vdash \sigma_m(s_1^{\mathrm{valid}}) \subsetneq s^{\mathrm{valid}}$: note that e describes the value of memory data in region s^{valid}, and e_1 describes the value of memory data in region s_1^{valid}, while in this case, the size of $\sigma_m(s_1^{\mathrm{valid}})$ is smaller than the size of s^{valid}. The size difference implies that $e = \sigma_m(e_1) \neq \top$ cannot be

true. Hence, there should be $\Delta \vdash \text{isNonChangeExp}(e) \wedge \sigma_m(e_1) = \top$. By discussing whether $e = \top$ and applying Sim-Val and Sim-Var-Map, we can derive $M'[\sigma(s^{\text{valid}}) + \delta_{\text{ptr}} + \delta_{\text{op}}] = M[\sigma(s^{\text{valid}})]$ from (9). In other words, for all $x \in \sigma(s^{\text{valid}})$, $M'[x+\delta_{\text{ptr}}+\delta_{\text{op}}] = M[x]$. Since $\sigma_1(s_1^{\text{valid}}) \subseteq \sigma(s^{\text{valid}})$ and $\delta_{\text{ptr1}} + \delta_{\text{op1}} = \delta_{\text{ptr}} + \delta_{\text{op}}$, then for all $x \in \sigma_1(s_1^{\text{valid}})$, $M'[x + \delta_{\text{ptr1}} + \delta_{\text{op1}}] = M[x]$. Therefore, the statement is true.

□

B.5 Public Noninterference

In this section, we prove that C transforms a well-formed program such that the output program satisfies public noninterference.

First, as shown in the following lemma, we prove that a well-formed program never writes secrets to the non-stack public region (s_{otherPub}).

Lemma 21. *If $P, \mathcal{P} \vdash_{TAL} S$, $S \xrightarrow{inst} S_1$, and inst stores (v, t) to address $v_a \in s_{otherPub}$, then $t = 0$.*

Proof. We prove for the case where $inst = \mathbf{movq}\, r_1, i_d(r_b, r_i, i_s)^{s,\tau}$. For other kinds of instructions that write to memory, it can be proved similarly.

Denote $S = (R, M, (pc, (\sigma_f, \sigma_b)); \Phi)$ and $\sigma = \sigma_f \cup \sigma_b$. According to store typing rules and well-formedness rules, $[v_a, v_a + 8) \subseteq \sigma(s)$. Mem-Type implies that either $\sigma(s) \subseteq s_{\text{otherPub}}$ or $\sigma(s) \cap s_{\text{otherPub}} = \emptyset$. Since $v_a \in s_{\text{otherPub}}$ and $v_a \in \sigma(s)$, $\sigma(s) \subseteq s_{\text{otherPub}}$.

Furthermore, denote $(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) = \text{getStateType}(\mathcal{P}, \text{nextPc}(pc))$. Then, $\mathcal{M}_1$ is the memory type after the store and $\mathcal{M}_1[s] = (\_, (\_, \tau))$ according to the store typing rules. According to Mem-Type, there should be $\sigma(\tau) = 0$. Dyn-Movq-r-m also constrains that $t \Rightarrow \sigma(\tau)$, so $t = 0$. □

Next, we prove the following lemma, which states that the transformed program never writes secrets to all public regions ($s_{\text{stackPub}} \cup s_{\text{otherPub}}$).

Lemma 22. *If $P, \mathcal{P} \vdash_{TAL} S$, $P, \mathcal{P} \vdash S' \prec S$, $S' \xrightarrow{C(inst)}{}^{*} S'_1$, and $C(inst)$ stores (v', t') to address $v'_a \in s_{stackPub} \cup s_{otherPub}$, then $t' = 0$.*

Proof. We prove for the case where $inst = \mathbf{movq}\, r_1, i_d(r_b, r_i, i_s)^{s,\tau}$. For other kinds of instructions that write to memory, it can be proved similarly.

Denote $S = (R, M, (pc, (\sigma_f, \sigma_b)); \Phi)$, $S' = (R', M', (pc', (\sigma'_f, \sigma'_b)); \Phi')$, $\sigma = \sigma_f \cup \sigma_b$, and $\sigma' = \sigma'_f \cup \sigma'_b$. Suppose *inst* stores (v, t) to address v_a. Then, Dyn-Movq-r-m implies that $t \Rightarrow \sigma(\tau)$, $R[r_1] = (v, t)$ and $R'[r_1] = (v', t')$. Since $P, \mathcal{P} \vdash S' \prec S$, then $t = t'$ according to Sim-Reg.

Furthermore, denote $(\Delta_1, \mathcal{R}_1, \mathcal{M}_1) = \text{getStateType}(\mathcal{P}, \text{nextPc}(pc))$ and $\omega = \text{getTransStrategy}(\mathcal{P}, \text{nextPc}(pc))$. According to the proof on the simulation relation for this **movq** instruction, there should be $\mathcal{M}_1[s] = (\_, (\_, \tau))$, $[v_a, v_a + 8) \subseteq \sigma(s)$ and $[v'_a, v'_a + 8) \subseteq \text{getShiftedSlot}(\mathcal{M}_1, \omega, \sigma, \sigma', s)$.

Mem-Type implies that $\sigma(s) \subseteq s_{\text{stackPub}} \vee \sigma(s) \subseteq s_{\text{otherPub}} \vee \sigma(s) \subseteq s_{\text{otherSec}}$ and

$$\begin{aligned} s' &:= \text{getShiftedSlot}(\mathcal{M}_1, \omega, \sigma, \sigma', s) \\ &= \sigma(s) + \text{getOpShift}(\mathcal{M}_1, \omega, s) + (\sigma'(\text{getPtr}(s)) - \sigma(\text{getPtr}(s))) \\ &= \sigma(s) + \text{getOpShift}(\mathcal{M}_1, \omega, s) + \text{getPtrShift}(\mathcal{M}_1, \omega, s). \end{aligned}$$

We consider the following cases:

- $\sigma(s) \subseteq s_{\text{stackPub}}$: According to Lemma 14, $s' = \sigma(s)$ or $\sigma(s) + \delta$. If $s' = \sigma(s)$ and $\omega(s) = \texttt{TransPtr}$, then

$$s' = \sigma(s) \Rightarrow \text{getPtrShift}(\mathcal{M}_1, \omega, \sigma, s) = 0 \Rightarrow \sigma(\tau) = 0 \Rightarrow t' = t = 0.$$

  If $s' = \sigma(s)$ and $\omega(s) = \texttt{TransOp}$, then

$$s' = \sigma(s) \Rightarrow \text{getOpShift}(\mathcal{M}_1, \omega, s) = 0 \Rightarrow \tau = 0 \Rightarrow t' = t = 0.$$

  If $s' = \sigma(s) + \delta$, then $s' \in s_{\text{stackSec}}$, which contradicts our assumption where $v'_a \in s'$ and $v'_a \in s_{\text{stackPub}} \cup s_{\text{otherPub}}$. Thus, we can ignore this case.
- $\sigma(s) \subseteq s_{\text{otherPub}}$: Suppose the store value of the original instruction *inst* is (v, t). According to Lemma 21, $t = 0$, so $t' = t = 0$.
- $\sigma(s) \subseteq s_{\text{otherSec}}$: According to Lemma 14, $s' = \sigma(s) \in s_{\text{otherSec}}$. Since $v'_a \in s'$, this contradicts our assumption that $v'_a \in s_{\text{stackPub}} \cup s_{\text{otherPub}}$, and we can ignore this case.

□

We then define public equivalence between two abstract machine states, which constrains that their taint bits represent whether their registers/memory slots share the same value. They should also have the same values in the public memory regions.

Definition 23 (Public Equivalence). We denote $(v, t) \simeq_{\text{taint}} (v', t')$ if $t = t' = 1 \vee (v = v' \wedge t = t' = 0)$. Then, states $S = (R, M, (pc, \_); \Phi)$ and $S' = (R', M', (pc', \_); \Phi)$ are publicly equivalent, i.e., $S \simeq_{\text{pub}} S'$ if

$$pc = pc' \qquad \forall r.\, R[r] \simeq_{\text{taint}} R'[r] \qquad \forall x.\, M[x] \simeq_{\text{taint}} M'[x]$$
$$\forall x \in s_{\text{stackPub}} \cup s_{\text{otherPub}}.\, M[x] = M'[x] = (\_, 0).$$

Finally, we can prove that the transformed program satisfies public noninterference, as shown in Lemma 24 and Theorem 26.

Lemma 24. *For all S_0, S_1, S'_0, S'_1 if $S'_0 \simeq_{pub} S'_1$ and for $i \in \{0, 1\}$,*

$$P, \mathcal{P} \vdash_{TAL} S_i \qquad P, \mathcal{P} \vdash S'_i \prec S_i,$$

then either S'_0, S'_1 are termination states or there exists S''_0, S''_1 such that

$$S'_i \xrightarrow{C(inst_i), o_i}{}^{*} S''_i, i \in \{0, 1\} \qquad o_1 = o_2 \qquad S''_0 \simeq_{pub} S''_1.$$

where o_i records all branch targets, load/store addresses, and store values to $s_{stackPub} \cup s_{otherPub}$ when executing $C(inst_i)$.

Proof. $S'_0 \simeq_{\text{pub}} S'_1$ implies that they share the same PC, so we can denote them as

$$S'_0 = (R_0, M_0, (pc, (\sigma_{f0}, \sigma_{b0})); \Phi_0) \quad S'_1 = (R_1, M_1, (pc, (\sigma_{f1}, \sigma_{b1})); \Phi_1).$$

If one of them is a termination state, the other must also be a termination state. If none of them is a termination state, then according to Theorem 11, there exist S''_0, S''_1 such that $S'_i \xrightarrow{C(inst), o_i}{}^{*}$

S_i'', $i \in \{0, 1\}$. Note that they execute the same instruction(s) since they share the same PC. Consider the following cases for *inst*:

Case movq $r, i_d(r_d, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}$**.** We have

$$C(inst) = \textbf{movq}\, r, \delta_0 + i_d(r_d, r_i, i_s)^{s_{\text{op}}+\delta_0, \tau_{\text{op}}}$$

According to Dyn-Movq-r-m and public equivalence,

$$R_0[r] = (v_1, t_1) \qquad R_1[r] = (v_2, t_2) \quad (v_1, t_1) \simeq_{\text{taint}} (v_2, t_2)$$
$$R_0[r_d] = R_1[r_d] = (v_d, 0) \quad R_0[r_i] = R_1[r_i] = (v_i, 0).$$

So S_0 and S_1 store to the same address $v_a := \delta_0 + v_d + v_i \times i_s + i_d$.

Furthermore, if $C(inst)$ stores to the public region, i.e., $v_a \in s_{\text{stackPub}} \cup s_{\text{otherPub}}$, then $t_1 = t_2 = 0$ according to Lemma 22. Thus, $(v_1, t_1) \simeq_{\text{taint}} (v_2, t_2)$ implies that $v_1 = v_2$. Therefore, $o_1 = o_2$. We can also easily derive that $S_0'' \simeq_{\text{pub}} S_1''$ by applying Dyn-Movq-r-m.

Case movq $i_d(r_d, r_i, i_s)^{s_{\text{op}}, \tau_{\text{op}}}, r$**.** We have

$$C(inst) = \textbf{movq}\, \delta_0 + i_d(r_d, r_i, i_s)^{s_{\text{op}}+\delta_0, \tau_{\text{op}}}, r$$

where $\delta_0 = 0$ or δ. Similarly, Dyn-Movq-m-r and public equivalence imply that both machines load from the same untainted address, so $o_1 = o_2$. $S_0'' \simeq_{\text{pub}} S_1''$ can also be derived by applying Dyn-Movq-r-m.

Case jne ℓ^{σ}**.** We have $C(inst) = inst$. According to Dyn-Jne and public equivalence, $R_0[\texttt{ZF}] = R_1[\texttt{ZF}] = (b, 0)$, which implies that both machines will jump to the same PC. Therefore, $o_1 = o_2$ and $S_0'' \simeq_{\text{pub}} S_1''$.

Case callq $f^{\sigma_{\text{call}}, \sigma_{\text{retq}}}$**.** We have

$$C(inst) = \textbf{addq}\, \delta, r_1; \ldots \textbf{addq}\, \delta, r_n; \textbf{callq}\, f^{\sigma'_{\text{call}}, \sigma_{\text{retq}}}.$$

Note that the **addq** instructions do not access memory or affect taint equivalence ($\simeq_{\text{taint}}$ relation) of register values. They also only update the PC to the next instruction. When executing **callq** $f^{\sigma'_{\text{call}}, \sigma_{\text{retq}}}$, according to Dyn-Callq, both machines will jump to f, i.e., the same PC. Furthermore, both machines store the same return address nextPC(P, pc) to the same untainted address. Therefore, the statement also holds for this case.

Case retq. We have $C(inst) = inst$. According to Dyn-Retq, both machines pop the return address from the same untainted address. The return address is also untainted and therefore the same for both machines. Therefore, the statement holds.

We omit other cases since the statement can be proved similarly. □

Definition 25 (Well-Formed Transformed State). S' is a well-formed state for $C(P)$ if there exists S such that $P, \mathcal{P} \vdash_{\text{TAL}} S$ and $P, \mathcal{P} \vdash S' \prec S$.

Theorem 26 (Public Noninterference). *If a program P is well-typed, then $C(P)$ satisfies software public noninterference.*

Proof. According to Definition 1, we just need to prove that for all well-formed initial states S_0 and S_1, if $S_0 \simeq_{\text{pub}} S_1$, then $[\![C(P)]\!]_{\text{pub}}(S_0) = [\![C(P)]\!]_{\text{pub}}(S_1)$ and $[\![C(P)]\!]_{\text{ct}}(S_0) = [\![C(P)]\!]_{\text{ct}}(S_1)$. This can be proved by applying Lemma 24 repeatedly. □

C BENCHMARK CHANGES

In `salsa20`, due to the limited heuristic rules in our valid-region-inference prototype, we explicitly initialize one stack-allocated array to zero. In `chacha20`, we add *SecSep* annotations derived from program semantics to facilitate inference by (1) identifying the trivial outcome of a complex initialization procedure and (2) encoding a memory nonoverlap assumption presented in the source code. We also modify `poly1305` to (1) eliminate the unsupported `goto`-based irreducible control flow and (2) avoid casting a structure to and from a byte array, which obscures the type and hinders type inference.

One can expect that as more heuristic rules are added and the type system becomes fully implemented, these manual efforts will also be eliminated.

D TOOLCHAIN EFFICIENCY

We profile our *SecSep* prototype by running the toolchain on all *SecSep*-annotated benchmarks using the test platform. The corresponding statistics are presented in Figure 17.

The total time required for type inference and checking ranges from 20 seconds to 35 minutes. The program transformation phase – omitted from the figure – relies exclusively on the inferred types and completes trivially fast, costing under one second for all benchmarks. While the toolchain may exhibit slower performance on large benchmarks, it yields a valuable and reliable model of binary-level information flow, enabling efficient execution and secure speculation even for complex cryptographic functions.

Both inference and checking spend over 90% of their time making numerous SMT solver calls, indicating a heavy reliance on the solver to guide heuristics and enforce type checking. *SecSep*'s performance can be further improved through optimizations such as tailoring SMT-solver invocations.

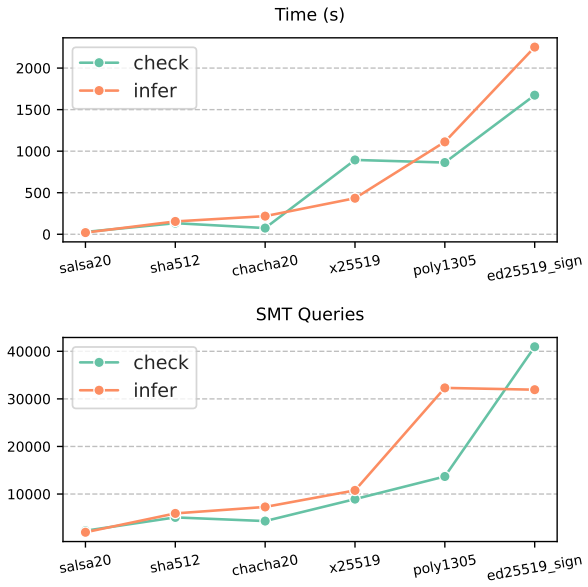


Figure 17: Efficiency of *SecSep* toolchain.