Securing Cryptographic Software via Typed Assembly Language

Shixin Song[*]
shixins@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Tingzhen Dong[*]
rogerdtz@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Kosi Nwabueze
kosinw@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Julian Zanders
jzanders@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Andres Erbsen
andreser@mit.edu
Google
Cambridge, United States

Adam Chlipala
adamc@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Mengjia Yan
mengjiay@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

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 → Side-channel analysis and countermeasures**; • **Theory of computation → Type structures**.

Keywords

Side-channel attacks and mitigation, information-flow security

ACM Reference Format:
Shixin Song, Tingzhen Dong, Kosi Nwabueze, Julian Zanders, Andres Erbsen, Adam Chlipala, and Mengjia Yan. 2025. Securing Cryptographic Software via Typed Assembly Language. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765116

1 Introduction

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

However, recent speculative-execution attacks, notably various Spectre attacks [34–36, 39, 51], 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 [9, 13] 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 [17, 19, 50, 58, 60, 65] 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

[*] Both authors contributed equally to this research.

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765116

design [19] 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 [19] and ConTExT [50], 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 stack variables 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 compilers, 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*, an assembly-transformation framework that partitions stack data securely.

We allow key information (usually lost during compilation) to be reintroduced in code through a new variant of typed assembly language (TAL), *Octal*, which facilitates sound program transformation. *Octal* is designed to enable *static* fine-grained taint tracking.

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 type annotations for function interfaces). 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 assembly programs based on their taint types. 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, they are 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 [10, 38]. We evaluate *SecSep*'s transformation with the hardware extension using six cryptographic benchmarks [25] 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 [55], provide a prototype implementation, and carry out an empirical evaluation.

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

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 [46, 61–63], TLBs [26], branch predictors [1, 22], and others [2, 5, 18, 21, 28, 29, 40, 48, 52, 57, 59]. 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, 51] 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) [24, 27, 41–43] was introduced to regain some of that information. We extend past results around memory safety to 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 coding discipline, in which the program 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 requirements.

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 [19] 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 use two observation models $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{pub}}$, following notations from prior work [31], to constrain our software requirements: constant-time and separating secret/public data. Specifically, $[\![\cdot]\!]$ 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, $[\![\cdot]\!]_{\text{ct}}$ records the trace of load/store addresses and branch targets, and $[\![\cdot]\!]_{\text{pub}}$ records a trace of data values stored in the public memory region. We then define public noninterference, the software property guaranteed by *SecSep*, where $S \simeq_{\text{pub}} S'$ constrains that two architectural states S and S' have equal values in the public memory region [55].

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 $[\![P]\!]_{\text{ct}}(S) = [\![P]\!]_{\text{ct}}(S')$ and $[\![P]\!]_{\text{pub}}(S) = [\![P]\!]_{\text{pub}}(S')$.

We use $\{\!|P|\!\}(S)$ to denote the microarchitectural observation trace of running program P on our out-of-order processor with initial state S [31]. The hardware must obey the following condition.

Definition 2 (Hardware Public Noninterference). A processor satisfies *hardware public noninterference* if for all programs 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 [19], 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

4.1 Limitations of Source-Level Annotation

A fundamental limitation of source-level code transformation used by prior works [19, 50] lies in its heavy reliance on assumptions about compiler internals. For example, to relocate secret stack variables, programmers must manually annotate their declarations with the section label `secret`, and the compiler is expected to allocate the variables in that section instead of the original stack. However, this strategy fails to guarantee public noninterference, due to 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 a cryptographic function `salsa20_words`. 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 pointer `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 11-12.

Given that array `x` holds secret data, a programmer can annotate it with the section label `secret`, 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 6-9 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 the 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 it to identify which instructions operate on secret data and need to be transformed. A further complication is the 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 under architectural constraints. Specifically, we may not use extra registers, which would require 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

```
1   void salsa20_words
2   (uint32_t *d, uint32_t s[16]) {
3     __attribute__
4     ((section("secret"))) static
5     uint32_t x[4][4];
6     int i;
7     for (i=0; i<16; ++i)
8       x[i/4][i%4] = s[i];
9   // Omit calculation on x
10  ...
11    for (i=0; i<16; ++i)
12      d[i] = x[i/4][i%4] + s[i];
13  }
```

(a) Annotated source code

```
1   salsa20_words:
2
3   # Public spill
4     movq %rdi, -64(%rsp)
5   # Source: line 7-8
6     movl (%rsi), %r12d
7     movl 4(%rsi), %r11d
8     movl 8(%rsi), %r9d
9     movl 12(%rsi), %eax
10  # Secret spill
11    movq %rax, -88(%rsp)
12  ...
```

(b) Original assembly code

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

(c) Transformed assembly code

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

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 taint type 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 memory operand with taint type 1. For example, in line 11, the 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.

The following sections 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 [43] 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.

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

Figure 2: *Octal* syntax

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**, **retq**, 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-typed 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).

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 $\mathcal{S}$, which serve as block preconditions.

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 a basic block (or more generally, an instruction sequence) ensures that the abstract machine whose state satisfies the type constraints can execute the block (instruction sequence) without getting stuck. When the machine is about to jump to another block at a branch instruction, its state should also satisfy the target's state type. Figures 3-5 elaborate with typing rules.

In general, each of the 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 initialized region in s ($s_{\text{addr}} \subseteq s^{\text{valid}}$) for memory safety, and the taint type of data in the slot must satisfy τ. *Octal* also requires the load address to be untainted.

Typing-Movq-r-m also constrains store instructions to be memory-safe and constant-time. It invokes Typing-StoreOp-Spill or Typing-StoreOp-Non-Spill depending on whether the store slot holds a spill or a data object in the source code. The difference arises from the different lifetimes of the two types of slots.

Typing-Movq-m-r

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

Typing-Movq-r-m

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

Typing-Cmpq-r-r

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

Typing-Jne

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

Typing-Callq

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

Figure 3: Instruction-sequence typing

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. Since each nonspill slot corresponds to a data object in the source file, we impose uniformity on the slot's taint type 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.

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

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

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

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

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

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

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

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

Figure 5: State subtyping

Typing-Jne specifies the rule for a conditional-branch instruction. *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 that might be changed by our transformation. 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 branch, 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.

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, detailed in [55].

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.

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 for the caller, 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 (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 formalize the type safety of *Octal* programs, that is, *Octal* guarantees well-typed programs to be executed on an *Octal* abstract machine without getting stuck. We define well-formedness of *Octal* abstract machine states as follows. 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. Then, the type safety is formalized using the following theorem. We provide details of the well-formedness definition and the proof of the type safety theorem in [55].

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. Instead, the correctness is checked separately by applying typing rules introduced in Section 5.

According to our type definitions, we need to generate state types of 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 typing information and exploring its implications.

6.1 Type initialization

We begin type inference by using type-unification variables to represent state types (i.e., $(\Delta, \mathcal{R}, \mathcal{M})$) and type annotations (i.e., load/store's destination-slot taint annotations; type-variable substitutions 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. To initialize memory typing $\mathcal{M}$, we first need to figure out $\text{dom}(\mathcal{M})$ for each function.

Determine Memory Layout. Core cryptographic routines usually do not allocate memory on the heap dynamically for reasons of performance, 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 arguments. Figure 6 provides an example using our annotation to describe a function argument (`mlen`) that gives the size of an array that another argument (`message`) points to. These annotations are compiled down to assembly and serve as specifications for functions. We also obtain the following information with simple compiler support: address ranges of function stack frames using a Clang pass, and locations of global variables indicated directly in assembly code. With the above support, our inference tool focuses on inferring type information for basic blocks within each function.

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

Figure 6: Example type annotations for function arguments 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.

Constraint-Movq-m-r-Unknown

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

Constraint-Movq-r-m-Unknown

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

Constraint-Jne

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

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. Note 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 apply to the range constraints with certain patterns and solve the predicates of each variable heuristically.

We primarily focus on inferring type variables used for pointer arithmetic, which is useful for reasoning about dependent types for load/store operations. Luckily, we target type inference for cryptographic programs, whose dependent-type range constraints share simple and intuitive patterns. Our empirical analysis found they follow the 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 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:

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

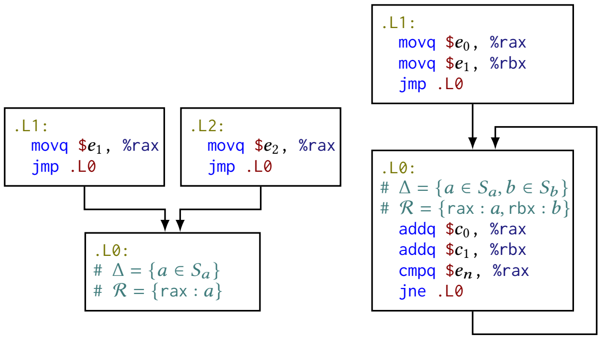
Infer range of loop counter. The second example (Figure 8b) shows the case where `rax` acts as a loop counter. `rax` is initialized to e_0 when entering the loop body from block `.L1` and increased by a constant step c_0 in each iteration. The loop ends when `rax` is equal to the boundary value e_n. 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 the range of loop counters.



```
.L1:
  movq $e_1, %rax
  jmp .L0
```

```
.L2:
  movq $e_2, %rax
  jmp .L0
```

```
.L0:
# Δ = {a ∈ S_a}
# R = {rax : a}
```

(a) Infer set of values

```
.L1:
  movq $e_0, %rax
  movq $e_1, %rbx
  jmp .L0
```

```
.L0:
# Δ = {a ∈ S_a, b ∈ S_b}
# R = {rax : a, rbx : b}
  addq $c_0, %rax
  addq $c_1, %rbx
  cmpq $e_n, %rax
  jne .L0
```

(b) Infer loop counter

Figure 8: Examples for dependent-type inference

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 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.

As a solution, we introduce another rule that infers the linear relation between type variables that share similar 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.

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

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.

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

```
foo:
# ℛ = {rdi : p, rsi : n}
# ℳ = {[p, p + n) : _}
  cmpq $8, %rsi
  jae .L0
  retq
.L0:
# Missing predicate:
# Δ = {n ≥ 8}
  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

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.

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

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 aspects: (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 constraint (2) and find the most accurate solution that covers the largest valid region to satisfy constraint (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 $\sigma_1(s^{\text{valid}}) \subseteq s_1^{\text{valid}}$.

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

Figure 10: Infer the valid region of an array

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.

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 each of their taint types as $\tau = x_1 \vee x_2 \vee \ldots x_n$. We can rewrite the constraint as $x_1 \Rightarrow 0 \wedge x_2 \Rightarrow 0 \wedge \ldots x_n \Rightarrow 0$.
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 $x_1 \vee x_2 \vee \cdots \vee x_n$ and the store operand's taint type as x_{op}. We can write the constraint as $x_1 \Rightarrow x_{\text{op}} \wedge x_2 \Rightarrow x_{\text{op}} \wedge \cdots \wedge x_n \Rightarrow x_{\text{op}}$.

In short, the taint constraints can be summarized in the form $E_1 \wedge E_2 \wedge \cdots \wedge E_n$. 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.

Here is how we derive taint predicates. 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 a 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 and illustrate how we apply these two strategies to transform programs. We then 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 switches 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. 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 inside the `memset` function 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. 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 as arguments 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 this pointer have the same taint type.[1]

Transform load/store operands. Next, our transformation uses a pass C_{op} to transform all load/store operands that access memory slots with transformation strategy `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, from the callee function's perspective, if a pointer argument references tainted slots with transformation strategy `TransPtr`, it should be shifted by δ by the caller at the call site, and no transformation is needed

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

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

(a) r12 → tainted

```
1  fparent_sec:
2    pushsecq %r12^{s+δ,τ}
3    movq $sec, %r12
4
5    callq child
6
7
8    popsecq %r12^{s+δ,τ}
```

(b) Secret r12

```
1  fparent_pub:
2    pushsecq %r12^{s+δ,τ}
3    movq $ptr, %r12
4    movq %r12, (%rsp)^{s,0}
5    callq child
6    movq (%rsp)^{s,0}, %r12
7    movq %rax, (%r12)
8    popsecq %r12^{s+δ,τ}
```

(c) Public r12

Figure 11: Restore callee-saved registers' taint

on the callee side. On the caller side, if the transformation strategy of the slots referenced by the same pointer is also TransPtr, we further propagate the transformation responsibility to the caller's call site. On the other hand, if the transformation strategy of those slots is TransOp, then the pointer is not transformed yet, so we need to add δ to the pointer argument when passing it to the callee.

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 has taint type $\tau \neq 0$ (i.e., it might be tainted). 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, restoring 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 call and retrieves them afterward. 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).

Note that although C_{callee} helps avoid unnecessary delays on speculation, it inserts extra instructions into the program and may cause performance overhead. We will evaluate this tradeoff by measuring the performance 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.*

Public Noninterference. Next, we briefly justify that the transformed program satisfies software public noninterference. 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.

Detailed formalization of the transformation and proof for the two properties can be found in [55], where we focus on the proof for major passes C_{op} and C_{ptr} and omit details for the optional pass C_{callee} that is relatively more straightforward.

8 Evaluation

8.1 Implementation and Experiment Setup

***SecSep* Toolchain.** We implement a prototype toolchain in OCaml, using Z3 [20] as the SMT solver. It includes (1) a parser for *SecSep*'s C source code annotations, (2) a parser for compiled x86-64 assembly programs, (3) type-inference rules and algorithms (Section 6), (4) a checker that validates inferred types against typing rules (Section 5.2), and (5) transformation based on inferred types (Section 7). This prototype is designed to cover the instructions in the benchmarks used for evaluation, and can be extended to support more instructions if needed. 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 [10, 38] replicating the defense idea from ProSpeCT [19]. Specifically, modules including ROB, register file, scheduler (InstructionQueue), load/store queue, and branch squash logic (Commit, IEW) are modified to support taint tracking and to delay transmitter instructions that leak secrets. We apply a microarchitecture configuration similar to that used in prior work on secure speculation [17]. 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

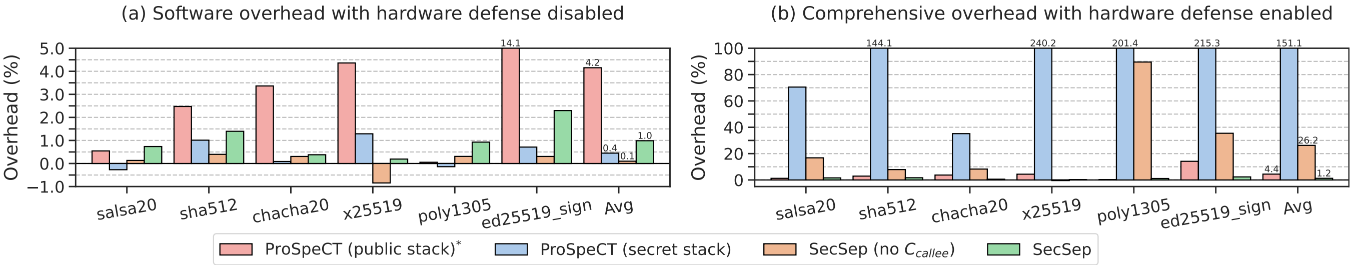


Figure 12: Execution-time overhead of transformed programs relative to original programs. * means the scheme is not secure.

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 from BoringSSL [25] (`sha512`, `chacha20`, `poly1305`, `x25519` and `ed25519_sign`). `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 [55].

We conduct our experiments on a test platform equipped with an Intel® Core™ i9-14900K CPU. Benchmarks are transformed using $\delta = -8$ MB and simulated 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 and relocate secret stack variables, while treating the original stack as public. Note that it cannot relocate secret stack spills and thus is insecure.
(2) ProSpeCT (secret stack): Manually relocate public stack variables while treating the original stack as secret. This approach conservatively protects any register spills and thus is secure.
(3) *SecSep* (no C_{callee}): Perform transformation passes C_{op} and C_{ptr}.
(4) *SecSep*: Perform all transformation passes, C_{op}, C_{ptr}, and C_{callee}.

Software Overhead. We compare the execution time of transformed programs running on unmodified hardware, scaled to the execution time of the original program, shown in Figure 12a. This is to understand 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 the defense, shown in Figure 12b. This is to understand how precisely each transformation separates secret/public data and the combined overhead introduced by software and hardware.

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 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 stack variables that need to be examined for correct annotation in ProSpeCT, and #Arg denotes the number of function arguments examined by *SecSep*. #Anno reports the number of lines of annotation.

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 secret/public data separation, thereby minimizing overtainting and enabling efficient execution when the defense is enabled. ProSpeCT (public stack) also achieves 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 shown 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, a significant overhead increase of 25% occurs when the hardware defense is enabled, highlighting the severity of overtainting caused by called routines. Therefore, despite adding extra instructions, C_{callee} is essential for efficient secure speculation. Hence, we include it as a key component in *SecSep*'s standard transformation scheme.

8.3 Manual Effort

We also compare the manual effort required by ProSpeCT and *SecSep* to annotate the source code for transformation as shown in Table 1. We focus on ProSpeCT's public-stack scheme, as it exhibits efficient execution and can be secure for certain benchmarks [19]. ProSpeCT requires examining stack variables in all functions (denoted as **F**), while *SecSep* only requires examining arguments of functions in the binary's call graph (denoted as **F'**). For fair comparison, we also count the stack variables and ProSpeCT annotations when examining only **F'**, with these numbers enclosed in parentheses.

Note the number of *SecSep* annotations grows linearly with the number of function arguments, because *SecSep* only requires identifying the memory layout and taint types of all function arguments for functions called during the execution of an application (**F'**). The annotation burst in `poly1305` is due to the frequent use of a 17-field structure as function arguments, while 14 of them share identical attributes and thus identical annotations. Writing *SecSep* annotations takes low effort since 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 restricted to **F'** when counting **#Var**. This pattern indicates that *SecSep* places less burden on the user by requesting only function-interface-level 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 effort 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 lift 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 [19] but 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 an argument 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 [24, 27, 41–43] 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, 54] 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 [25].

Software mitigations against Spectre. Several prior works [6, 8, 16, 44, 45, 47, 53, 54, 56, 66] 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) [12]. Many of these approaches [16, 44, 45, 47, 56, 66] introduce large performance overhead since they unavoidably block safe speculation conservatively when blocking insecure speculation. Other works [6, 8, 53, 54] managed to achieve speculative noninterference with marginal overhead by applying SLH intelligently, but they are built upon research-prototype source languages such as FaCT [15] or Jasmin [3, 4], thereby not compatible with off-the-shelf cryptographic libraries such as BoringSSL [25]. Furthermore, many of them [8, 16, 47, 53, 54, 56, 66] only consider speculation at conditional branches in their speculative control-flow analysis, so they cannot prevent leakage introduced by other speculation primitives [14, 30, 32, 35, 36, 49].

Hardware mitigations against Spectre. Prior works [7, 17, 37, 58, 64, 65] adopt hardware taint tracking and delay speculative operations that transmit secrets. These pure hardware solutions require no software changes but face challenges in identifying secret data in memory. STT [65] and others [7, 58, 64] only consider speculatively loaded data as secrets, while leaving nonspeculatively loaded data unprotected, thereby not guaranteeing constant time [31]. SPT [17] 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 [19] and others [23, 50] propose to make the software separate secret and public data into coarse-grained regions so that the hardware can easily identify secrets. Our paper provides an assembly-rewriting approach to generating programs satisfying this requirement, which offers more accurate separation and requires less manual effort.

Compiler approach to separate secret and public data. ConfLLVM [11] 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.

Acknowledgments

The authors thank the Matcha Group (MIT) for their help and the anonymous CCS reviewers for their feedback. This work was supported in part by a gift from Amazon; by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-22-1-0511; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

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] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 785–800. doi:10.1145/3319535.3363194

[10] 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. doi:10.1145/2024716.2024718

[11] 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.

[12] Chandler Carruth. n.d.. *Speculative load hardening*. https://llvm.org/docs/SpeculativeLoadHardening.html

[13] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new Spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. doi:10.1145/3385412.3385970

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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 *30th IEEE Symposium on Security and Privacy*. IEEE, 45–60.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[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, Jerónimo Castrillón, 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 Mück, 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 Éder 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, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software* (Atlanta, GA, USA). 25–35.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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*.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Shixin Song, Tingzhen Dong, Kosi Nwabueze, Julian Zanders, Andres Erbsen, Adam Chlipala, and Mengjia Yan. 2025. Securing cryptographic software via typed assembly language (extended version). arXiv:2509.08727 [cs.CR] https://arxiv.org/abs/2509.08727

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.