Type Checker for Annotated Assembly Programs

by

Julian Zanders

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Julian Zanders
Department of Electrical Engineering and Computer Science
May 9, 2025

Certified by: Adam Chlipala
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Type Checker for Annotated Assembly Programs

by

Julian Zanders

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The rise of speculative-execution attacks, such as Spectre, has presented a security challenge to developers. Speculation on secret data can expose it, but running without speculation is suboptimal for runtime. To fix this, researchers have been evaluating "smart" speculation schemes, which determine when to speculate and when not to in order to balance runtime with security.

Our lab proposes Octal, a solution that utilizes software and hardware in tandem. Data values are marked as secret or public using type inference, and the veracity of inference is checked using a type checker. Then, hardware can separate the secret and public values.

My contributions were to the type checker, as well as some scripting to evaluate results.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

Many thanks to my supervisor, Adam Chlipala, as well as PhD students Shixin Song and Tingzhen Dong, for their guidance and support throughout the process!

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Speculative Execution

Early CPU optimization focused on physical characteristics of the chip, such as size and clock frequency. However, as physical limitations have made this increasingly unviable over time, designers have shifted their attention to optimizing the instruction pipeline [1]. Of the pipeline optimizations that have been implemented, speculative execution has perhaps had the most prominent implications on computer security.

Speculative execution is closely related to **out-of-order processing**, which is the non-sequential execution of program instructions. When implemented correctly, out-of-order processing does not violate functional correctness. For example, if two `add` instructions execute on entirely disjoint pairs of registers, it doesn't actually matter which of the two instructions executes first. The architectural state will end up in the same state regardless.

CPUs that can perform out-of-order processing can then be extended with branch predictors to implement **speculative execution**. Upon reaching a branch, speculative execution has the CPU **speculate** on the most likely result of the branch condition and continue executing instructions on the corresponding control-flow path. If the speculation is correct, then the program ultimately runs faster, since it did not have to stall any instructions while awaiting the conditional result. Modern branch predictors use heuristics to speculate correctly far more often than not. However, as we will see in a future section, these heuristics can also be exploited by attackers.

It is important to note that preserving a program's functional correctness requires that only the instructions on the correct control flow path are committed to the architectural state. Therefore, speculative execution depends on **transient execution**, which occurs when the CPU executes instructions and stores them in the microarchitectural state but does not immediately commit those results to the architectural state. On correct branch predictions, the results must be committed from the microarchitecture to the architecture. Otherwise, the results must be squashed.

```
1     if(x < sensitiveArray.length) {
2       int y = sensitiveArray[x];
3       int z = instrumentArray[y * cacheLineSize];
4     }
```

Figure 1.1: Example of Code Vulnerable to Speculative Execution

1.2 Spectre Attack

Unfortunately for computer security, preserving functional correctness enforces no constraints on the microarchitectural state. This has left most CPUs vulnerable to speculative execution attacks, the most notable of which for our purposes is called "**Spectre**" [1].

Spectre and its variants leverage the fact that, while an otherwise secure computer program may not leak any secrets to the architectural state, its underlying microarchitectural implementation may store secrets that can be indirectly observed. For example, suppose that a software program implemented a conditional branch that, if true, would load contents of a secret memory location. Even if the condition evaluates to false during runtime, if the condition were predicted to be true, the secret data may well be loaded into the microarchitecture, such as in the cache.

It is not possible for a developer to directly read the contents of cache memory. However, by using **covert channels**, information about the cache state can be inferred. For example, in the code snippet in Figure 1.1, suppose that the branch predictor incorrectly guessed that the branch would be taken, causing lines 2 and 3 to be speculatively executed. Now, even if an attacker doesn't have direct access to `sensitiveArray`, they can still exploit the fact that the contents of `instrumentArray` were loaded into the cache during speculative execution. By looping over the values in `instrumentArray` and determining which access takes the least time, the attacker can infer that that value corresponds to the one preloaded into the cache. As such, the contents of `sensitiveArray` have been leaked [1].

1.3 Speculative Constant Time

Researchers have iterated on a number of software mitigations for speculative-execution attacks. There is **Cryptographic Constant Time (CCT)**, which requires that software be written without using secret data for variable-time operations (most notably branch conditions and memory accesses) [2]. This helps mitigate timing side channels, but since speculative execution exploits microarchitectural features, CCT is not sufficient as a defense against attacks like Spectre.

As such, researchers have moved on to a stricter property called **Speculative Constant Time (SCT)**. A program satisfies SCT if for any two initial states with identical public values, there are no observable differences in the machine's architectural or microarchitectural state after running any series of directives [2]. (Note that this property inherently satisfies CCT.) This generalized statement of SCT covers any arbitrary side channel, including those created by speculative execution.

Chapter 2

Introduction

Our project aims to automatically transform programs satisfying Cryptographic Constant Time into programs that satisfy Speculative Constant Time without violating speculative noninterference. Our approach utilizes software and hardware in tandem, with each easing the job of the other.

Previous hardware-only SCT implementations have either required heavy hardware changes or have not preserved the Speculative Noninterference Property [3, 4]. Previous hardware-software SCT implementations have required a prohibitive amount of manual transformation on the original programs [5]. We aim to extend the existing research to achieve Speculative Constant Time, achieve Speculative Noninterference, minimize hardware changes, and minimize manual effort.

2.1 Implementation

The lab has developed a hardware-software solution known as Octal.

2.1.1 Hardware and Software

In terms of hardware, it is relatively easy to track taint status directly in a register, and it is relatively easy to track taint status on page level. Octal follows this model. On the architectural level, each register in an Octal machine is augmented with a bit indicating taint status, whereas system memory is preemptively partitioned into public and secret regions. On a software level, an x86 Assembly source program can then be automatically rewritten such that memory accesses involving secrets are addressed to the secret region and memory accesses involving public data are addressed to the public region.

2.1.2 Program Transformation: Type Inference

Octal relies on inferring the basic types of registers and memory, as well as inferring arithmetic relations among those type variables. These basic types are comprised of the dependent type, which represents the data value; and the taint status, which represents whether or not the value should be treated as secret. Thus, a type can be instantiated for each possible architectural state of the program, and subtyping relations among different architectural states can be defined.

2.1.3 Program Transformation: Type Checking

Type checking is the process of verifying that a term will eventually evaluate to a value (i.e. an nonstuck term in normal form). In the context of Octal, type checking uses the inferred type of each basic block as input. It then checks that, at each taken jump instruction, the type representing the state of the program when the jump is taken is a subtype of the inferred architectural state of the jump target.

2.1.4 My Contributions

My main contribution was implementing symbolic execution for the type checker. I also annotated benchmarks to compare our approach to ProSpeCT [5].

Chapter 3

Symbolic Execution for Type Checking

3.1 Technologies

Symbolic execution was implemented using a number of preexisting software solutions.

- **Microsoft Z3.** Microsoft Z3 is a Satisfiability Modulo Theories (SMT) solver. SMT solvers attempt to determine whether or not a given set of constraints are satisfiable. There are a number of such "theories" corresponding to the type of the variables in the constraints. For example, there are theories on bit vectors, integers, Booleans, and so on. Since our basic types are comprised of dependent types, which represent binary numbers stored in the computer architecture, and taint types, which represent a binary taint status, we used a combination of the bit vector theory and the Boolean theory to implement our checker.

- **OCaml.** OCaml is a functional programming language. Its strong typing paired with the functional paradigm make it very well-suited for problems in program verification.

3.2 Type Representation

Before anything else, the type of the architectural state must be defined.

3.2.1 Architectural State Representation

We represent the architectural state of our abstract machine as a compound type. Among other things, this compound type consists of a context, a register file, a memory map, and a collection of flags.

The register file is a record type that maps register names to basic types. The memory map is a record type that maps memory regions to basic types. The flag map is a record type that maps flags to Booleans. A basic type is a tuple of one dependent type (representing the value being stored) and one taint type (representing the taint status of that value).

$$
\begin{array}{rcc}
b & := & 0 \mid 1 \\
e & := & x \mid v \mid \top \mid e_1 \oplus e_2 \mid \ominus e \\
\tau & := & x \mid 0 \mid 1 \mid \tau_1 \vee \tau_2 \\
\beta & := & (e, \tau) \\
R & := & \{r_1 : \beta_1, ...\} \\
M & := & \{m_1 : \beta_1, ...\} \\
F & := & \{f_{carry} : b_{carry}, f_{sign} : f_{sign}, ...\} \\
A & := & (R, M, F)
\end{array}
$$

Figure 3.1: Simplified Type Representation of the Architectural State

3.2.2 Dependent-Type Representation

Dependent types are represented in Z3 as bit vectors. Like most values in Z3, bit vectors can be represented as both interpreted constants (bit vectors with known values) and uninterpreted constants (bit vectors with unknown values). Regardless of whether or not they are interpreted, arithmetic expressions on those bit vectors can be built with Z3's standard bit vector operations.This enables us to represent program state symbolically even with unknown values.

3.2.3 Taint-Type Representation

Taint types are represented in Z3 as Booleans, where true (1) corresponds to tainted and false (0) corresponds to untainted. Just as with bit vectors, Z3 can represent taint types with both interpreted and uninterpreted values.

3.2.4 Architectural-State Representation

Figure 3.1 displays a partial description of the architectural state type. It is not comprehensive but rather defines the aspects of the state type that are relevant to symbolic execution.

3.3 Implementation

The type checker's symbolic execution was implemented using the Microsoft Z3 OCaml library. This was effectively an implementation of small-step semantics for each x86 Assembly instruction that appeared in our benchmarks.

3.3.1 Control Flow

The type checker is static, so it only ever runs instructions sequentially. The instruction itself determines how the architectural state will be updated. However, conditional jump instructions cannot be handled as in normal program execution, as it is not generally possible to evaluate the condition statically. Instead, for jump instructions, we must generate Z3 checks, and then assert constraints on the architectural state for subsequent instructions in the basic block. This process is as follows:

When a conditional jump instruction is reached (with `jmp` being equivalent to a conditional jump whose condition is simply `true`)...

1. Push a new scope onto the Z3 solver.
2. Assert that the branch condition is true.
3. Generate a Z3 check that verifies that the current architectural state is a subtype of the branch target.
4. Pop the scope off of the Z3 solver.
5. Assert that the branch condition is false.
6. Proceed to the next instruction.

This process allows for both branch results to be statically tested. The taken check is asserted under the assumption that the static type meets the branch condition, and then symbolic execution proceeds under the assumption that the static type does not meet the branch condition.

3.3.2 Small Steps on Instructions

Overview

Our OCaml implementation subdivides instructions into unary, binary, and ternary. For each of these instructions, the operands and any needed flag statuses are passed as inputs into the symbolic-execution function. The outputs of this function are the resulting basic type (comprised of a dependent type and a taint status) and the updated flag map.

For arithmetic and bitwise instructions such as `add`, `mul`, and `xor`, dependent types are computed directly using functions built into the Z3 library. Similarly, for taint tracking on these instructions, taint types are computed using Z3 library functions.

Symbolic execution for some instructions requires more complex computations than for others. For example, certain instructions only use a certain subset of the bits in a register, requiring intermediate masking or extraction steps to compute accurately. An in-depth discussion of typing rules for each instruction can be found in Appendix A.

The result of the symbolic execution is then used to update the correct architectural elements and generate a new state. That new state is used for the next instruction, and the cycle repeats.

$$A : (R, M, F) \to A' : (R', M', F')$$

```
(declare-const rax!1 (_ BitVec 64))
(declare-const fcarry!1 Bool)
(declare-const fparity!1 Bool)
(declare-const faux!1 Bool)
(declare-const fzero!1 Bool)
(declare-const fsign!1 Bool)
(declare-const foverflow!1 Bool)
(assert (= rax!1 (bvadd rax!0 #x0000000000000008)))
(assert (= fcarry!1 (not (bvadd_no_overflow rax!0 #x0000000000000008))))
...
```

Figure 3.2: Static Single-Assignment for `addq %rax $0x08`.

Static Single-Assignment

We have seen that the small step can be represented as the generation of a new state. However, in practice, full state representations are not generated for each instruction. Instead, for each instruction, new interpreted constants are generated for each modified architectural element.

For example, Figure 3.2 displays the Z3 code generated for the instruction `addq %rax, $0x08`. This will affect the accumulator as well as six flags of interest. Thus, we declare new constants for those elements and constrain their values according to the instruction.

3.3.3 Iterations on Implementation

In the process of developing the type checker, I went through a number of iterations on my codebase. The most informative iterations were on our memory representation.

Memory Representation

Representing memory slots in Z3 was one of the most challenging problems. A register can be trivially represented by creating a new Z3 constant, as a (bit vector, Boolean) tuple can easily represent the dependent type and taint type. However, for memory, it is not nearly as straightforward. Memory is one contiguous block that assembly code can address into at any region, and each memory address may contain its own value and taint status.

My first attempt was to maintain a Z3 array. In Z3, arrays are not "arrays" in the traditional sense; they are far more analogous to dictionaries or hash tables. The user is free to define both the type of the key and the type of the value. It seemed, then, that Z3 was well-equipped to model memory as a bit vector → (bit vector, Boolean) array. The memory address could be the key, and the basic type the value. This seemed to have the advantage of letting arbitrary symbolic expressions on bit vectors index into the array, which would provide natural support for x86 instructions like `lea`.

Unfortunately, this approach ended up being prohibitive in terms of runtime. While Z3 was able to successfully type check some tiny toy programs with this approach, maintaining the array was prohibitively expensive, with Z3 unable to return any result even after many hours of runtime.

Ultimately, the approach that ended up working was precomputing memory slots via type inference and giving a single Z3 constant for each slot. A slot is defined by the range of addresses that it spans, and then the full slot has a single basic type. Determining which instructions operate on which memory slot can then be delegated to type inference.

Chapter 4

Evaluating Octal on Benchmarks

To evaluate the user-friendliness of Octal, we wanted to compare the manual effort of generating annotated assembly to the manual effort of annotating a C program. Manual C code annotation is used in other secure speculation projects, such as ProSpeCT [5], so it serves as a suitable baseline for comparison.

ProSpeCT requires that all secret variables be stored in their own memory region. Practically, this can be achieved in C code by marking secret variables as `static` and storing them in a custom memory region with attribute markers.

We apply this annotation style to implementations of three representative cryptographic functions: salsa20 (a stream cipher), sha512 (a cryptographic hash function), and ed25519 (a digital signature scheme). The implementations were provided in C by the BoringSSL library. Each secret value was marked as such, and values that were used to compute those secrets were also marked as secret. Arrays were also marked as secret if they were not used as indices or pointers, as they commonly serve as instruments in Spectre-like attacks. All other values were marked as public.

Since Octal aims to separate secret and public data into separate memory regions, developers would theoretically have the choice of either moving secret data into static memory and leaving public data on the stack, or moving public data into static memory and leaving secret memory on the stack. As such, we attempted both options for each benchmark and evaluated our results.

I annotated each of the three aforementioned benchmarks and wrote a small Python script (which can be found in Appendix B) to count the number of manual annotations. This allows us to visualize the amount of manual work that Octal saves over other solutions.

The results can be seen in Figure 4.1. This admittedly small sample would suggest that annotating secret data tends to require more manual annotation than annotating public data. However, the amount of work saved on the part of a human programmer when annotating only public variables would be negligible, as they would still have to identify all of the secret variable declarations as such to decide *not* to annotate them. As such, the potential for human error (either by omitting necessary annotations or incorrectly annotating a declaration) with so many required annotations is evident.

Benchmark	Public Declarations	Secret Declarations	Total Annotations
Salsa20	5	7	12
SHA512	8	7	15
ED25519	11	231	242

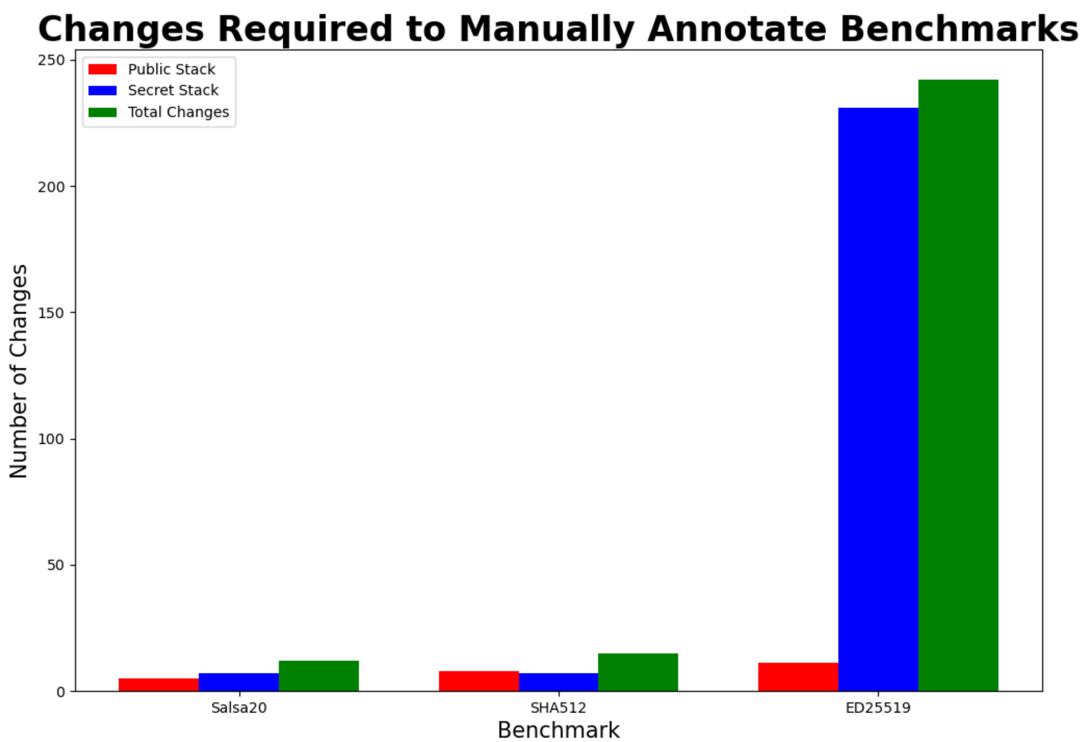

Figure 4.1: Number of Manual Annotations Required for Each Benchmark

Appendix A

Typing Rules

This appendix contains a description of the typing rules implemented by symbolic execution.

A.1 Architectural State Type

Below is a description of the architectural state type A.

$$\begin{array}{rcc} b & := & 0 \mid 1 \\ e & := & x \mid v \mid \top \mid e_1 \oplus e_2 \mid \ominus e \\ \tau & := & x \mid 0 \mid 1 \mid \tau_1 \vee \tau_2 \\ \beta & := & (e, \tau) \\ R & := & \{r_1 : \beta_1, \ldots\} \\ M & := & \{m_1 : \beta_1, \ldots\} \\ F & := & \{f_{carry} : b, f_{sign} : b, ...\} \\ A & := & (R, M, F) \end{array}$$

We can also define types of operations. Our approach divided operations by the numbers of operands they take as input. Additionally, all operations take the flag map as input.

$$\begin{array}{rcc} uop & := & \beta \to F \to (\beta, F) \\ bop & := & \beta \to \beta \to F \to (\beta, F) \\ top & := & \beta \to \beta \to \beta \to F \to (\beta, F) \end{array}$$

From there, we can define our small-step semantics. For example, we can define small steps for unary instructions whose results are stored in registers, or binary instructions whose results are stored in memory slots.

Small Step (Unary Instruction, Register Destination)

$$\frac{a : (R_a, M_a, F_a) \qquad op : uop \qquad src_1 : \beta \qquad flags : F \qquad op(src_1, flags) = (\beta', F') \qquad dst \in R_a}{a, op \rightarrow (R_a[dst \rightarrow \beta'], M_a, F')}$$

Small Step (Binary Instruction, Memory Destination)

$$\frac{a : (R_a, M_a, F_a) \qquad op : bop \qquad src_1 : \beta_1 \qquad src_2 : \beta_2 \qquad flags : F \qquad op(src_1, src_2, flags) = (\beta', F') \qquad dst \in M_a}{a, op \rightarrow (R_a, M_a[dst \rightarrow \beta'], F')}$$

A.2 Rules for Instructions

Here, we will define typing rules for a representative set of instructions. For ease of expression, we will first define the following set of auxiliary functions:

- Let BW be a function that returns the bitwidth of the given dependent type. For example, $BW(10101010) = 8$.
- Let EXT be a function that extracts the specified slice of bits (inclusive) from the given dependent type. For example, $EXT(110101, 3, 1) = 010$.
- Let SB be a function that returns the number of set bits in the passed dependent type. For example, $SB(10010010) = 3$.
- Let MSB be a function that returns the most significant bit of the passed dependent type. For example, $MSB(1000000000000000) = 1$.
- Let BT be a function that takes two dependent types representing a bit string and a position and returns the bit at that position in the bit string. For example, $BT(11101, 1) = 0$.
- Let SE be a function that sign-extends the given dependent type by the given number of bits. For example, $SE(111, 3) = 111111$ and $SE(011, 3) = 000011$.
- Note that we assume that each of the arithmetic operations $+, -, \times,$ and $/$ outputs a bit vector with the minimum-length bitwidth to hold the full result, while the modulus operator (%) is assumed to truncate its input to the minimum-length bitwidth to hold the full result. For example, $1111 + 1 = 10000$, whereas $(1111 + 1)\%2^4 = 0000$. This distinction will be relevant for a number of typing rules for both dependent types and flags.

Arithmetic Operations

The arithmetic operations `add`, `adc`, `sub`, and `sbb` have nearly identical typing rules. They set the carry, parity, auxiliary, zero, sign, and overflow flags according to their results. `inc` and `dec` use nearly identical typing rules as well, with the caveat that they preserve the state of the carry flag and use that carry flag as the second operand. Below are full typing rules for `add` and `adc`, from which the rules for the other aforementioned instructions can be extrapolated.

ADD src_1, src_2

$$\frac{src_1 : (e_1, \tau_1) \qquad src_2 : (e_2, \tau_2) \qquad fl : F \qquad e_{result} = e_1 + e_2 \qquad e_{dest} = e_{result} \% 2^{BW(e_1)}}{add(src_1, src_2, fl) \rightarrow \left((e_{dest}, \tau_1 \vee \tau_2), fl \begin{bmatrix} f_{carry} & \rightarrow & e_{result} >= 2^{BW(e_{dest})} \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0)) \% 2 == 0 \\ f_{aux} & \rightarrow & EXT(e_1, 3, 0) + EXT(e_2, 3, 0) \geq 2^4 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & \neg(-2^{BW(e_{dest})-1} \leq e_{dest} < 2^{BW(e_{dest})-1}) \end{bmatrix} \right)}$$

ADC src_1, src_2

$$\frac{\begin{array}{c} src_1 : (e_1, \tau_1) \qquad src_2 : (e_2, \tau_2) \qquad fl : F \qquad e_{add} = e_2 + fl[f_{carry}] \\ e_{result} = e_1 + e_{add} \qquad e_{dest} = e_{result} \% 2^{BW(e_1)} \end{array}}{adc(src_1, src_2, fl) \rightarrow \left((e_{dest}, \tau_1 \vee \tau_2), fl \begin{bmatrix} f_{carry} & \rightarrow & e_{result} >= 2^{BW(e_{dest})} \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0)) \% 2 == 0 \\ f_{aux} & \rightarrow & EXT(e_1, 3, 0) + EXT(e_{add}, 3, 0) \geq 2^4 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & \neg(-2^{BW(e_{dest})-1} \leq e_{dest} < 2^{BW(e_{dest})-1}) \end{bmatrix} \right)}$$

The multiplication instructions `mul` and `imul` are different in that they set fewer flags (those being the carry and overflow flags). `imul` in particular also has varied behavior depending on the number of arguments given to the instruction. Note that both `mul` and the one-operand form of `imul` implicitly take the bottom $BW(e_{src})$ bits of the accumulator as an input operand. Also note that the three-operand form of `imul` only varies in having a specified destination rather than an implied destination, so the two variants get identical rules.

MUL src

$$\frac{src : (e_{src}, \tau_{src}) \qquad acc : (e_{acc}, \tau_{acc}) \qquad fl : F \qquad e_{result} = e_{src} \times e_{acc} \qquad e_{dest} = e_{result} \% 2^{2 \times BW(e_{src})}}{mul(src, acc, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{acc}), fl \begin{bmatrix} f_{carry} & \rightarrow & (e_{dest} \gg BW(e_{src})) \neq 0 \\ f_{overflow} & \rightarrow & (e_{dest} \gg BW(e_{src})) \neq 0 \end{bmatrix} \right)}$$

IMUL src (ONE SOURCE OPERAND)

$$\frac{src : (e_{src}, \tau_{src}) \qquad acc : (e_{acc}, \tau_{acc}) \qquad fl : F \qquad e_{result} = e_{src} \times e_{acc} \qquad e_{dest} = e_{result} \% 2^{2 \times BW(e_{src})}}{imul(src, acc, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{acc}), fl \begin{bmatrix} f_{carry} & \rightarrow & e_{dest} == EXT(e_{result}, BW(e_{src}) - 1, 0) \\ f_{overflow} & \rightarrow & e_{dest} == EXT(e_{result}, BW(e_{src}) - 1, 0) \end{bmatrix} \right)}$$

IMUL src_1, src_2 (TWO/THREE SOURCE OPERANDS)

$$\frac{src_1 : (e_1, \tau_1) \qquad src_2 : (e_2, \tau_2) \qquad fl : F \qquad e_{result} = e_1 \times e_2 \qquad e_{dest} = e_{result} \% 2^{BW(e_1)}}{imul(src_1, src_2, fl) \rightarrow \left((e_{dest}, \tau_1 \vee \tau_2), fl \begin{bmatrix} f_{carry} & \rightarrow & e_{dest} == EXT(e_{result}, BW(e_{dest}) - 1, 0) \\ f_{overflow} & \rightarrow & e_{dest} == EXT(e_{result}, BW(e_{dest}) - 1, 0) \end{bmatrix} \right)}$$

Bitshift Operations

Since flags are set differently depending on the value of the second operand, binary bitshift operations tend to have relatively convoluted typing rules. For simplicity, we can write entirely distinct rules based on the value of the second operand. We will also assume that the second operand evaluates to a number that does not equal or exceed the bitwidth of the first operand. (If we did not make this assumption, we would have to either address masking effects or model undefined x86 behaviors in our typing rules.) Typing rules for bit rotation instructions `rol` and `ror` will not be explicitly defined, but a similar ruleset covers those instructions.

The left logical and arithmetic shifts are identical instructions.

$$\textsc{sal/shl}\ src, cnt\ (\textsc{Count} = 0)\quad \frac{src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad e_{cnt} = 0 \qquad fl : F}{shl(src, cnt, fl) \rightarrow ((e_{src}, \tau_{src} \vee \tau_{cnt}), fl)}$$

$$\textsc{sal/shl}\ src, cnt\ (\textsc{Count} = 1)\quad \frac{\begin{array}{c} src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad e_{cnt} = 1 \qquad fl : F \\ e_{dest} = e_{src} \ll e_{cnt} \qquad f'_{carry} = BT(e_{src}, BW(e_{src}) - e_{cnt}) \end{array}}{shl(src, cnt, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{cnt}), fl \begin{bmatrix} f_{carry} & \rightarrow & f'_{carry} \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & MSB(e_{dest}) \neq f'_{carry} \end{bmatrix} \right)}$$

$$\textsc{sal/shl}\ src, cnt\ (\textsc{Count} > 1)\quad \frac{src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad 1 < e_{cnt} < BW(e_{src}) \qquad fl : F \qquad e_{dest} = e_{src} \ll e_{cnt}}{shl(src, cnt, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{cnt}), fl \begin{bmatrix} f_{carry} & \rightarrow & BT(e_{src}, BW(e_{src}) - e_{cnt}) \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \end{bmatrix} \right)}$$

By constrast, logical right shift and arithmetic right shift are different operations. We will cover logical right shift as a representative instruction, but note that the arithmetic right shift sets the overflow flag differently on one-bit shifts.

$$\textsc{shr}\ src, cnt\ (\textsc{Count} = 0)\quad \frac{src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad e_{cnt} = 0 \qquad fl : F}{shr(src, cnt, fl) \rightarrow ((e_{src}, \tau_{src} \vee \tau_{cnt}), fl)}$$

$$\textsc{shr}\ src, cnt\ (\textsc{Count} = 1)\quad \frac{src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad e_{cnt} = 1 \qquad fl : F \qquad e_{dest} = e_{src} \gg e_{cnt}}{shr(src, cnt, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{cnt}), fl \begin{bmatrix} f_{carry} & \rightarrow & BT(e_{src}, e_{cnt} - 1) \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & MSB(e_{src}) \end{bmatrix} \right)}$$

$$\textsc{shr}\ src, cnt\ (\textsc{Count} > 1)$$

$$\frac{src : (e_{src}, \tau_{src}) \qquad cnt : (e_{cnt}, \tau_{cnt}) \qquad 1 < e_{cnt} < BW(e_{src}) \qquad fl : F \qquad e_{dest} = e_{src} \gg e_{cnt}}{shr(src, cnt, fl) \rightarrow \left((e_{dest}, \tau_{src} \vee \tau_{cnt}), fl \begin{bmatrix} f_{carry} & \rightarrow & BT(e_{src}, e_{cnt} - 1) \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & MSB(e_{src}) \end{bmatrix} \right)}$$

Also not explicitly defined here, but supported by symbolic execution, are the `shld` and `shrd` instructions. Their definition is comparable to the left- and right-shift instructions described above. The `bswap` instruction is also supported, which swaps the endianness of the dependent type but leaves taint status and flags unaffected.

Bitwise Operations

Bitwise operations `and`, `or`, and `xor` all have nearly identical typing rules. They set the flags identically, with the only difference being the bitwise operation itself. We will look at `and` as a representative instruction.

$$\textsc{and}\ src_1, src_2$$

$$\frac{src_1 : (e_1, \tau_1) \qquad src_2 : (e_2, \tau_2) \qquad fl : F \qquad e_{dest} = e_{src1} \wedge e_{src2}}{and(src_1, src_2, fl) \rightarrow \left((e_{dest}, \tau_1 \vee \tau_2), fl \begin{bmatrix} f_{carry} & \rightarrow & 0 \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & 0 \end{bmatrix} \right)}$$

There are also unary bitwise operations `not` and `neg`.

$$\textsc{not}\ src$$

$$\frac{src : (e_{src}, \tau_{src}) \qquad fl : F}{not(src, fl) \rightarrow ((!e_{src}, \tau_{src}), fl)}$$

$$\textsc{neg}\ src$$

$$\frac{src : (e_{src}, \tau_{src}) \qquad fl : F \qquad e_{dest} = -e_{src}}{neg(src, fl) \rightarrow \left((e_{dest}, \tau_{src}), fl \begin{bmatrix} f_{carry} & \rightarrow & e_{src} \neq 0 \\ f_{parity} & \rightarrow & SB(EXT(e_{dest}, 7, 0))\%2 == 0 \\ f_{aux} & \rightarrow & EXT(e_{dest}, 3, 0) < 0 \\ f_{zero} & \rightarrow & e_{dest} == 0 \\ f_{sign} & \rightarrow & MSB(e_{dest}) \\ f_{overflow} & \rightarrow & \neg(-2^{BW(e_{dest})-1} \leq e_{dest} < 2^{BW(e_{dest})-1}) \end{bmatrix} \right)}$$

Move Operations

The `mov`, `movz`, and `movs` instructions are considered unary, as the value of the input is simply copied to the destination with the appropriate type of bit extension. Below is the typing rule for `mov`, which represents all three well.

$$\dfrac{\textsc{mov}\ dst, src \quad src : (e_{src}, \tau_{src}) \qquad fl : F}{mov(src, fl) \rightarrow ((e_{src}, \tau_{src}), fl)}$$

In the context of Octal's type checker, `lea` is implemented identically to `mov`. (When the source operand is passed to the checker, the memory slot has already been resolved by type inference.)

This leaves `cmoveq` as the only binary move operation.

$$\textsc{cmoveq}\ dst, src \quad \dfrac{src : (e_{src}, \tau_{src}) \qquad dst : (e_{dst}, \tau_{dst}) \qquad fl : F}{cmoveq(dst, src, fl) \rightarrow (\text{if } fl[f_{zero}] \text{ then } (e_{src}, \tau_{src}) \text{ else } (e_{dst}, \tau_{dst}), fl)}$$

Miscellaneous Instructions

The `bt` instruction is unique in that it does not affect any register or memory slot. Instead, it only affects the carry flag. In the context of the type checker, the operation still returns a basic type as a destination, but the destination will ultimately be discarded. Thus, we simply use a dummy basic type.

$$\textsc{bt}\ pos, str \quad \dfrac{pos : (e_{pos}, \tau_{pos}) \qquad str : (e_{str}, \tau_{str}) \qquad fl : F}{bt(pos, str, fl) \rightarrow \big((0, \tau_{pos} \vee \tau_{str}), fl\left[\ f_{carry} \quad \rightarrow \quad BT(e_{str}, e_{pos})\ \right]\big)}$$

The remaining instructions covered by Octal are packed floating-point operations: `punpck` `packxs`, `padd`, `psub`, `pxor`, `pandn`, `pand`, `por`, `psll`, `psrl`, `xorp`, and `pshuf`. While these operations do appear in the benchmarks, they appear very infrequently and have little meaningful effect on the final results. Thus, in this iteration of Octal, symbolic execution evaluates these operations to a dependent type of $\top$ and ignores the flags.

$$\textsc{Pack Instruction} \quad \dfrac{src : (e_{src}, \tau_{src}) \qquad dst : (e_{dst}, \tau_{dst}) \qquad fl : F}{pack(dst, src, fl) \rightarrow ((\top, \tau_{src} \vee \tau_{dst}), fl)}$$

Appendix B

Benchmark Evaluation Script

Here is the Python code used to evaluate benchmarks after annotation.

```python
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import sys
import argparse
import os

script_dir = Path(__file__).resolve().parent
proj_dir = script_dir.parent
src_dir = proj_dir / "src"


class Edit:
    def __init__(self, original, edited, line_number):
        self.original = original
        self.edited = edited
        self.line_number = line_number

    def __str__(self):
        return f"{self.line_number}: {self.edited}"

#########################
# FUNCTIONAL PARAMETERS #
#########################

"""
These functions implement some of the key functionality of the
   comparison routine.
Editing these will edit the way the comparator evaluates its lines.
"""


def pub_stack_search(base_line: str, edit_line: str):
```

```python
    """
    Returns whether or not the `edit_line` contains a secret section
        attribute.
    This only counts towards the total if the two lines are not
        identical.
    """
    return "PUBLIC_VAR" in edit_line and base_line != edit_line


def sec_stack_search(base_line: str, edit_line: str):
    """
    Returns whether or not the `edit_line` contains a public section
        attribute.
    This only counts towards the total if the two lines are not
        identical.
    """
    return "SECRET_VAR" in edit_line and base_line != edit_line


def trim_line(inp: str, commenting: bool):
    """
    Returns the result of removing all comments and trailing/leading
        whitespace
    from the inputted line of C source code.
    Also returns whether or not the following line is part of a multi
        -line comment.
    """
    result = inp

    # Follow multi-line comments
    if commenting:
        if (end_comment_idx := inp.find("*/")) == -1:
            return "", True
        result = inp[end_comment_idx+2:]

    # Remove trailing comments
    next_multiline = result.find("/*")
    next_inline = result.find("//")
    first_comment = min([next_multiline, next_inline], key=lambda x:x
        if x != -1 else float("inf"))
    multiline_commenting = False
    if first_comment != -1:
        result = result[:first_comment]
        multiline_commenting = first_comment == next_multiline

    # Purge #include statements and calls to Valgrind
    if "#include" in result or "valgrind" in result.lower():
```

```python
        result = ""

    return result.strip(), multiline_commenting


##########################
# SOURCE CODE COMPARATOR #
##########################

"""
These functions implement the source code comparator.
"""

def compare_lines(base_src: list, edit_src: list, compare_func):
    """
    Given the lines of the base source code and the lines of the edited
    source code, returns the list of all edits made between the base
    and edited code. It is assumed that all edits are:
    (1) made inline OR
    (2) edit whitespace OR
    (3) edit comments

    Other types of edits will break the comparator, so don't do those.
    """
    def find_next(src: list, ptr: int, commenting: bool):
        line = None
        while ptr < len(src):
            line, commenting = trim_line(src[ptr], commenting)

            if len(line) != 0:
                break

            ptr += 1

        return line, ptr, commenting

    all_edits = []

    base_ptr, base_commenting = 0, False
    edit_ptr, edit_commenting = 0, False
    while True:
        base_line, base_ptr, base_commenting = find_next(base_src, base_ptr, base_commenting)
        edit_line, edit_ptr, edit_commenting = find_next(edit_src, edit_ptr, edit_commenting)
```

```python

        if base_ptr == len(base_src) or edit_ptr == len(edit_src):
            break

        if compare_func(base_line, edit_line):
            all_edits.append(Edit(base_line, edit_line, base_ptr))

        base_ptr += 1
        edit_ptr += 1

    return all_edits


def analyze_one_set(base_file: Path, pub_file: Path, sec_file: Path):
    """
    Analyzes one set of files for changes among the three.
    A "set" of file consists of the base source file,
    the source file modified to place all public variables on the stack,
    and the source file modified to place all secret variables on the stack.

    Returns a tuple of two lists, where the lists are the edits in the
    public and secret stack variants, respectively.
    """

    # Read all three files
    with base_file.open() as f:
        base_contents = f.readlines()
    with pub_file.open() as f:
        pub_contents = f.readlines()
    with sec_file.open() as f:
        sec_contents = f.readlines()

    pub_edits = compare_lines(base_contents, pub_contents, pub_stack_search)
    sec_edits = compare_lines(base_contents, sec_contents, sec_stack_search)

    print(f"Found {len(pub_edits)} variables marked public")
    print(f"Found {len(sec_edits)} variables marked secret")

    return pub_edits, sec_edits


def generate_fileset(directory: Path, base_name: str):
```

```python
    """
    Given the directory to the source files (relative to src_dir)
    and the name of the base file (NOT including the .c extension),
    returns a tuple of the three files in the set.
    """

    file_dir = src_dir / directory
    return (
        file_dir / f"{base_name}.c",
        file_dir / f"{base_name}_stack.c",
        file_dir / f"{base_name}_stack.c"
    )


def plot_results(results, labels, save_dir):
    """
    Plots the results of the experiments in a cute little bar chart.
    """
    bar_width = 0.25
    fig = plt.subplots(figsize=(12, 8))

    publics = [len(result[0]) for result in results]
    secrets = [len(result[1]) for result in results]
    totals = [pub + sec for pub, sec in zip(publics, secrets)]

    bars1 = np.arange(len(publics))
    bars2 = [position + bar_width for position in bars1]
    bars3 = [position + bar_width for position in bars2]

    plt.bar(bars1, publics, color="r", width=bar_width, label="Public
        Stack")
    plt.bar(bars2, secrets, color="b", width=bar_width, label="Secret
        Stack")
    plt.bar(bars3, totals,  color="g", width=bar_width, label="Total
       Changes")

    plt.title("Changes Required to Manually Initialize Benchmarks",
       fontweight="bold", fontsize=24)
    plt.xlabel("Benchmark", fontsize=15)
    plt.ylabel("Number of Changes", fontsize=15)
    plt.xticks([r + bar_width for r in range(len(publics))], labels)

    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    plt.savefig(save_dir / "changes.png")

```

```
    plt.legend()
    plt.show()


if __name__ == "__main__":
    fileset_data = [
        ("salsa20", "standalone_salsa20", "salsa20"),
    ]

    filesets = [generate_fileset(folder, name) for folder, name, _ in
        fileset_data]
    results = [analyze_one_set(base, pub, sec) for base, pub, sec in
       filesets]

    parser = argparse.ArgumentParser()
    parser.add_argument("-p", "--plot", help="Set to true to plot
       results")
    args = parser.parse_args()

    if args.plot:
        plot_results(results,
                    [data[2] for data in fileset_data],
                    script_dir / "analysis")
```

References

[1] C. Canella, J. V. Bulk, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *28th USENIX Security Symposium* (2019). URL: https://www.usenix.org/system/files/sec19-canella.pdf.

[2] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. “Constant-Time Foundations for the New Spectre Era”. In: *PLDI ’20’* (2020). DOI: 10.1145/3385412.3385970.

[3] B. C. Pierce. *Software Foundations*. Vol. 7. Cambridge, UK: Princeton University, 1920.

[4] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data”. In: *MICRO ’52’* (2019). URL: https://iacoma.cs.uiuc.edu/iacoma-papers/micro19_2.pdf.

[5] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens. “ProSpeCT: Provably Secure Speculation for the Constant-Time”. In: (2023). DOI: 10.48550/arXiv.2302.12108.