Reasoning about Control Flow in the Presence of Transient Faults

Frances Perry and David Walker
Princeton University
{frances, dpw}@cs.princeton.edu

Princeton University Technical Report TR-799-07

Abstract

A transient fault is a temporary, one-time event that causes a change in state or erroneous signal transfer in a digital circuit. These faults do not cause permanent damage, but when they strike conventional processors, they may result in incorrect program execution. While detecting and correcting faults in first-order data may be accomplished relatively easily by adding redundancy, protecting against faults during control flow transfers is substantially more difficult. This paper analyzes the problem of maintaining the control-flow integrity of a program in the face of transient faults from a formal theoretical perspective. More specifically, we augment the operational semantics of an idealized assembly language with additional rules that model erroneous control-flow transfers. Next, we explain a strategy for detecting control-flow errors based on previous work by Oh [11] and Reis [16]. In order to reason about the correctness of the strategy relative to our fault model, we develop a new assembly-level type system designed to guarantee that any control flow transfer to an incorrect block will be caught before control leaves that block. The key technical result of the paper is a rigorous proof of this fundamental control-flow property for well-typed programs. We also prove that this new typed assembly language is sufficiently expressive to serve as a target for type-preserving compilation from a simple language of while programs.

1 Introduction

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [3, 19, 10], rendering processors that use them more susceptible to *transient faults*. Transient faults or *soft errors* are often caused by external events, such as an energetic particle striking silicon atoms within a chip. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values.

While transient faults are currently rare, they have already been noticed in commodity processors and have caused significant failures. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and caused crashes in server systems at major customer sites, including America Online, eBay, and dozens of others [4]. More recently at the Los Alamos Neutron Science Center, Hewlett Packard acknowledged their AlphaServer ES45 supercomputer was frequently crashing due to soft errors [7].

More importantly, as each processor generation increases clock rates, lowers voltages and increases the density of transistors, transient faults are more likely to occur. Figure 1, which is taken from work by Shenkar Borkar [5], illustrates the current and projected future trends in transient fault rates relative to chip feature size. With current trends suggesting fault rates are increasing at approximately 8% per chip generation, we may see a 100% rise in transient faults in just seven years.

In order to counter the future threat of transient faults, researchers from industry and academia have been searching for solutions to the reliability problem in both hardware and software. Broadly speaking, with

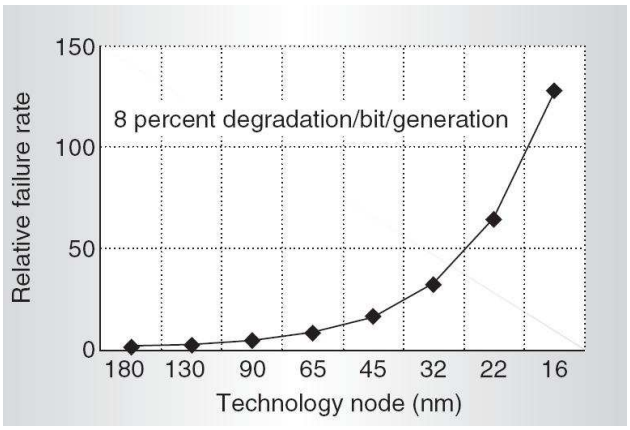

Figure 1: Transient Fault Rates vs. Feature Size

sufficient hardware resources, hardware-only solutions are more efficient for a single, fixed reliability policy, but software-only solutions are more flexible and less costly. In terms of flexibility, software-only solutions may be deployed *immediately* on current hardware that already exists in the field, simply by recompiling the application in question. Consequently, if Los Alamos Labs is repeatedly suffering from soft errors because of insufficient protection in their current supercomputing facilities, software solutions offer the hope of an effective way to solve their problem right now. In terms of cost-effectiveness, recent studies have shown that software techniques for fault tolerance often add approximately 35% overhead [16] to the computation with no additional hardware cost, whereas a standard double- or triple- modular redundancy technique will add 100% or 200% to the hardware cost, with some additional performance overhead for communication between replicas. Hence, depending upon where a given application sits in the cost-performance-reliability trade-off space, software, hardware or some mix of the two may be the preferred solution.

Unfortunately, devising software solutions to the problem of transient faults, and making sure they are correct, is an extremely difficult task. Just as the many possible interleavings of threads make it difficult to reason about the properties of concurrent programs, the many possible scenarios in which transient faults can arise make it difficult to reason about the properties of faulty programs. Moreover, just as conventional testing is often an ineffective way to uncover bugs in concurrent programs, testing is likely to be an ineffective way to uncover reliability errors in possibly faulty programs.

Faced with these challenges, we and other researchers at Princeton have recently begun to develop type-theoretic techniques for reasoning about software in the presence of transient faults. In our first effort [21], we devised a lambda calculus called λ_{zap} to serve as a highly idealized model for unreliable computations. The operational semantics of the calculus specify that any value may suddenly be corrupted during execution. However, programs are able to replicate computations and use atomic voting operations to check replicas against one another to detect and recover from transient faults. A type system for λ_{zap} guarantees that any well-typed program is fault tolerant. In our second piece of work [13], we studied fault tolerance in the more realistic setting of assembly language with specialized hardware instructions to aid detection of faults. Once again we devised a type system (this time called TAL_{FT}) and rigorously proved that it guarantees a strong fault tolerance property for all well-typed programs. From a theoretical perspective, these type systems codify formal reasoning techniques that allow programmers to prove strong reliability properties of their programs. Equally importantly, from a practical perspective, these type systems can be implemented and used to check the correctness of compiler outputs. Using a type checker to verify these reliability properties, where possible, is vastly superior to conventional testing as the type checker gives *perfect coverage* relative to the fault model whereas any test suite will be highly incomplete.

Despite the progress made to date, this prior work skirts the issue of how to reason about code that not only incurs faults to first-order data, but also may go wrong during a control flow transfer. The faulty

lambda calculus λ_{zap} avoids the issue altogether by assuming the existence of high-level atomic operations to simultaneously check for errors, recover and jump to a new control flow point. The fault-tolerant typed assembly language TAL_{FT} admits the possibility of faults to the program counter, but requires a highly specialized instruction set and additional nonstandard hardware state to detect those faults.

Surprisingly, however, researchers [11, 16] have developed techniques for detecting certain classes of control-flow errors entirely in software. As mentioned above, software techniques have an advantage over hardware in that they may be deployed selectively and immediately at low cost to solve problems that arise in the field. Unfortunately, none of these new techniques have been proven sound. Oh *et al.* [11] evaluate the effectiveness of their techniques empirically, showing the number of harmful faults decreases by an order of magnitude, but give no specification of exactly which faults they attempt or do not attempt to detect. Reis *et al.* [16] lay out in careful English exactly which faults they believe their system defends against and where they believe the vulnerabilities remain. However, they give no mathematical specification of the semantics of their target machine, their fault model or their desired properties, and they make no attempt to prove that their claims are correct.

Though these techniques are promising, many theoretical questions remain. In particular, is it possible to characterize the effectiveness of these techniques *analytically* as opposed empirically? In other words, can we prove that such techniques are sound with respect to an interesting and non-trivial, though incomplete, fault model? One of the key benefits of such an analysis is that it would guarantee an important fragment of the problem has been thoroughly solved and thereby free researchers to study auxiliary instrumentation techniques that address the remaining incompleteness. Perhaps more importantly, the formal fault model would define an important hardware/software interface: The software has been proven to handle faults that lie within the model; future hardware designers need only provide mechanisms to catch those faults that lie outside the model. While this latter point may appear of little importance since TAL_{FT} already demonstrates how to design a sound hybrid hardware-software protection system, the key difference is that such results would show how to shift a substantial portion of the control-flow checking burden from the hardware to software. This may lead to much simpler hardware designs as well as the opportunity to trade performance for reliability at *compile time* as opposed to *hardware design time*.

In this paper we attack these theoretical questions following a similar strategy to our earlier work. First, we define an incomplete, yet simple, elegant and non-trivial control-flow fault model — one in which faults can cause jump instructions and conditional branches to transfer control to the beginning of any program block. Next, we develop a type system that guarantees a strong fault tolerance property relative to this fault model. We have proven our type system is sound and also have demonstrated that it is sufficiently expressive that we can compile a classic while programs into well-typed programs in the language.

To summarize, there are three main contributions of this research.

- We have defined the first procedure (a type system) for verification of assembly code in the presence of transient control-flow faults. From a technical perspective, the type system introduces a novel way of classifying the reliability properties of program values and entire machine states, generalizing the earlier "color systems" used by λ_{zap} and TAL_{FT} . The type system is also of interest for the way it uses a collection of abstract types to track the state of the fault tolerance protocol.

- We have formulated a powerful fault-tolerance theorem for our type system and have developed the proof techniques that allow us to validate it. The key technical challenge we overcome is the fact that after a control-flow fault has occurred, it is impossible to count on almost *any* standard program invariant. So, how can one carry out a proof of type preservation under such circumstances?

- We have demonstrated that our type system is suitably expressive by showing how to compile a simple language of while programs into well-typed assembly language. We prove our translation is type-preserving.

The rest of the paper explains the problem of transient control flow faults and our techniques for reasoning about them in more detail. First, Section 2 gives additional intuition about the problem and solution by explaining a simple assembly-language protocol for detecting control-flow errors. This protocol is a simplified

version of the protocols used by Oh [11] and Reis [16]. Section 3 begins the more formal work by defining the syntax and operational semantics of an idealized assembly language. It also shows how to model erroneous control-flow transfers by adding special rules to the standard operational semantics. Section 4 defines the type system that guarantees that assembly code follows the simple protocol outlined earlier in Section 2. This type system, particularly the special value and machine state typing rules, codify the major invariants needed to prove type safety and the subsequent strong reliability properties. Section 5 sketches the major components of the fault-tolerance proof. Section 6 shows that our typed assembly language is sufficiently expressive that it is possible to translate while programs into well-typed, fault-tolerant code. Finally, Section 7 discusses further related work and Section 8 concludes. Throughout the paper we provide proof sketches of the key lemmas and theorems. The complete proofs are provided in an online appendix [14].

2 Informal Overview

When a transient fault causes the actual sequence of control flow blocks visited by a program to deviate from the expected sequence, we say a control-flow error has occurred. In this paper, control-flow errors arise in three different ways: (1) there may be a fault to the target address of a jump instruction; (2) there may be a fault to the target address of a conditional jump instruction; or (3) there may be a fault to the boolean used to decide whether to jump or fall through a conditional. Such faults may occur immediately prior to attempting the control-flow transfer or at any other time during the computation. However, whenever a control-flow operation is executed, we assume execution is either transferred to the beginning of some valid block, or to some invalid block or illegal instruction. In the latter case, we assume the hardware immediately catches an attempt to execute the illegal instruction. We do not consider the possibility that a fault causes a control flow transfer to a legal instruction in the middle of some valid block.

As is standard, we adhere to the *Single Event Upset* model [15, 17], which states that only one fault may occur during an execution. However, even though just one fault occurs, faulty values may be copied, propagated and used in any way an ordinary value may be used. Hence, a single fault can lead to arbitrarily many corrupted values if not caught soon after it occurs.

The goal of this work is to develop and prove correct a software protocol that guarantees such control-flow errors can never go undetected. The central challenge in this endeavor is to overcome the problem that *no single value can ever be trusted to be correct* — a transient fault may strike any value in any register. Consequently, as is usual in fault tolerance, the solution is to avoid relying on any single value by replicating the critical state and checking replicas against one another. In this case, the critical state is the value of the program counter. Checking the correctness of a control-flow transfer involves creating a replica of the intended control-flow destination and then checking the replica against the real program counter to detect any difference.

To be more specific, compiled code creates the replica prior to any control-flow transfer by moving the intended destination into a designated register. We refer to this register as the *intentions register* `ri`. This intentions register is part of the global "calling convention" for fault-tolerant control flow transfers. We fix the register so that all jump targets know where to find the intended destination, even when there has been a control-flow fault.

As an example, to jump to address `L2`, one might use the following code sequence. In this code, we leave ellipsis in between instructions to emphasize our system allows flexible scheduling of instructions — ordinary instructions may be interleaved with the instructions used to guarantee fault tolerance.

```
L1: ...
    movi ri, L2
    ...
    movi r2, L2
    ...
    jmp r2
```

Since the intentions register `ri` plays a special role in the protocol for detecting control-flow errors, we will need to type check the move instruction that loads this register in a special way. To designate the move as special, we henceforth write it `intend L2` rather than `movi ri, L2` as in the following example code.

```
L1: ...
    intend L2
    ...
    movi r2, L2
    ...
    jmp r2
```

If the intentions register has been set properly prior to all jump instructions, the jump targets are able to catch control flow errors. To be specific, all jump targets should be instrumented with the following code.

```
Lk: movi r2, Lk
    ...
    sub r2, r2, ri
    ...
    brz r2, lrecover
    ...
```

Here, the current block address `Lk` is loaded into some register `r2`. That register is then compared with the contents of `ri` and if there is any difference, control is transferred to `lrecover`, an address containing recovery code.[1] Once again, since the branch to the recovery code plays a special role in the fault-tolerance protocol, we give it the special syntax `recovernz r2`. Thus, our detection code will henceforth be written as follows.

```
L2: movi r2, L2
    ...
    sub r2, r2, ri
    ...
    recovernz r2
    ...
```

As an example of how a transient fault might be caught using our protocol, suppose register `r2` is corrupted just prior to attempting to execute the jump to `L2` in block `L1`. Upon arrival at some erroneous control flow block, say `L3`, the intended destination `L2` remains safely untouched in register `ri`, though, unnervingly, all other program invariants may be disrupted. The target code compares the contents of `ri` (*i.e.,* `L2`) with `L3`, which it loaded into `r2` after arriving at the current block. It detects a difference and jumps to the recovery code.

One must also consider what happens if faults strike at different times or in different places. For instance, the jump target might have been corrupted much earlier than we suggested above, perhaps just after being initially loaded into `r2`, instead of just prior to the jump. Will that make a difference? In this case, no. Likewise, `ri` might be corrupted, either before or after jumping. In this case, we reach the correct destination, but it appears as though there was a fault because `ri` differs from the current block label (assuming the fault occurs prior to the subtraction). Unable to tell the difference between a fault in the intentions register and a fault in the control-flow transfer itself, we jump to recovery code.[2] A number of other scenarios must also be analyzed — in order to have confidence in the solution, one must do so in a principled, disciplined fashion.

[1] Since recovery is a secondary issue to detection, we do not consider it in this paper.

[2] Adding a third redundant piece of state would make it possible to differentiate between a real control-flow error and a fault to `ri`. It would also make recovery relatively easy. However, we content ourselves with a relatively simple fault-detection-only scheme in this paper.

It is important to observe that similar, but subtly different code sequences do not adequately protect against faults. In particular, optimizations like copy propagation, common subexpression elimination and some code motion transformations, are no longer semantics-preserving in the context of transient faults. For instance, the following simple change to the way block `L1` was written above leads to a vulnerability.

```
L1: movi r2, L2   (*)
    ...
    movi ri, r2   (**)
    ...
    jmp r2
```

Here, a single transient fault to `r2` anywhere between execution of instructions `(*)` and `(**)` results in an uncaught control-flow fault as both the jump target and the intentions register will simultaneously be incorrect.

Likewise, the code motion transformation illustrated below shifts the move from a target block into the jumping block and creates a vulnerability.

```
L1: movi r2, L2
    intend L2
    movi r3, L2
    jmp r2

Lk: sub r3, r3, ri  (***)
    recovernz r3
    ...
```

Above, a fault to `r2` causes a control-flow error, but testing `r3` against `ri` at line `(***)` will not help detect the fault. The conclusions to draw from these examples are that the correctness properties of this code are indeed subtle and that verifying fault tolerance properties *after* the compiler has completed its suite of performance optimizations may help detect errors in code generation.

Conditional Branches. The protocol for handling conditional branches is slightly more involved than the case for jumps, but follows a similar pattern. We begin by assuming that the condition for the jump is held in registers `r4` and `r4'`. These two registers must be *independent replicas* of one another. In other words, in the absence of faults, they should contain the same boolean value, and moreover, a fault to one should have no impact on the value of the other. Given this assumption (which will be verified by our type system), the following code sequence sets up a conditional branch, which may fall through to `L2` or may jump to `L3`. The code uses a conditional branch `brz r4, r3`, which jumps to `r3` if `r4` is zero and otherwise falls through to `L2`. It also uses a conditional move `cmovz r4', ri, r3'`, which moves the contents of `r3'` into `ri` if `r4'` is zero, and otherwise does nothing. [3]

```
L1: ...
    // assumes r4 and r4' are independent replicas
    ...
    movi r3, L3
    movi r3', L3
    ...
    intend L2
    cmovz r4', ri, r3'
```

[3]Many architectures including the IA-32 following the Pentium Pro, the Sparc V-9 and the IA-64 have conditional moves. If the architecture does not have a a conditional move, a conditional branch and a move instruction can be used instead, but this branch will not be protected against faults.

$$
\begin{array}{llll}
\textit{colors} & c & ::= & G \mid B \mid O \\
\textit{colored values} & v & ::= & c\ n \\
 & & & \\
\textit{code memory} & C & ::= & \cdot \mid C, \ell \to b \\
 & & & \\
\textit{registers} & r & ::= & r_i \mid r_1 \mid \ldots \mid r_n \\
\textit{register file} & R & ::= & \cdot \mid R, r \to v \\
 & & & \\
\textit{history} & h & ::= & \ell_1, \ldots, \ell_n \\
\textit{instructions} & i & ::= & \texttt{movi}\ r_d\ v \mid \texttt{sub}\ r_d\ r_s\ r_s \\
 & & & \mid \texttt{intend}\ r_t \mid \texttt{intendz}\ r_z\ r_t \\
 & & & \mid \texttt{recovernz}\ r_z \\
\textit{blocks} & b & ::= & i; b \mid \texttt{jmp}\ r_t \mid \texttt{brz}\ r_z\ r_t \\
 & & & \\
\textit{state} & \Sigma & ::= & (C, h, R, b) \\
\textit{final states} & \mathcal{F} & ::= & \Sigma \mid \texttt{recover}(h) \mid \texttt{hwerror}(h)
\end{array}
$$

Figure 2: Machine State Syntax.

```
    brz r4, r3
L2: ...
    ...
L3: ...
```

Again, to notate the special role of `ri` and simplify the presentation, we will henceforth write the conditional move `cmovz r4', ri, r3'` as `intendz r4', r3'`. Intuitively, the `intend` instruction unconditionally sets the intentions register, whereas the `intendz` instruction conditionally sets the intentions register. The error-detection code in blocks labeled `L2` and `L3` is identical to the error-detection code discussed earlier for jumps, as it must be.

Summary With just a few, well-thought-through instructions, it is possible to create a redundant copy of the intended destination of any control flow transfer prior to initiating the transfer itself. Moreover, at any control-flow target, it is possible to use that redundant copy to check that code has actually arrived at the proper place. However, as our examples illustrated, it is also easy to make slight errors in the process. In addition, since transient faults can occur at so many different places in the protocol and influence so many different bits of state, one needs proof to believe such a protocol will work. Hence, in the following sections, we make the machine's operational semantics and fault model precise and develop a sound type system strong enough to verify that the "good" instruction sequences we have discussed in this section are indeed fault tolerant.

3 The Control-Flow Machine

For clarity and elegance, we will work with a minimal assembly instruction set involving move (`movi`), subtraction (`sub`), jump (`jmp`) and conditional branch if zero (`brz`) instructions as well as the special macros `intend`, `intendz` and `recovernz`.[4] Instruction operands include constant values v and registers r. In the previous section, values were written unannotated, but from this point forward we annotate every value with a *color* (either green G, blue B or orange O). These colors have no operational significance, but they play

[4]The examples in the previous section assumed `intend` took a constant argument, whereas our calculus specifies it takes a register argument. If the reader find this disturbing, the examples can easily be rewritten to insert a move instruction prior to the `intend`. Alternatively, with little work, we could add a constant-argument `intend` instruction to the calculus.

a special role in the type system and proof of correctness. The only kind of value is an integer. In general, meta-variable n ranges over integers, but when we wish to emphasize that an integer will be used as an address, we use the meta-variable ℓ.

Instructions are grouped together in code blocks b. These blocks are always terminated by either a jump or a conditional branch instruction. Code memory C is a partial map from addresses to valid code blocks b. Addresses are ordered, and we use the notation $\ell + 1$ to refer to the address of the block following the block at ℓ. If a block at ℓ ends with a conditional branch, we assume $\ell + 1$ inhabits the domain of C — in other words, conditional branches always have a block to fall through to.

The register file R is a mapping from registers to the colored values they contain. The registers include the intentions register r_i and a number of general-purpose registers r_1 through r_n. We use the notation $R(r)$ to denote the contents of r in R. We use the notation $R[r \mapsto v]$ to denote a new register file R' created by updating R so it maps r to v. When we wish to refer to the unannotated integer n as opposed to the colored value $c\ n$ in a register r in R, we use the notation $R_{val}(r)$. Similarly, $R_{col}(r)$ refers to the color annotating the value in r.

An ordinary abstract machine state Σ is a tuple containing code C, history h, register file R and code block to be executed b. The history h is a sequence of labels. It records the code blocks visited during the current execution. In addition to ordinary abstract machine states, there are two special "final states." The state $\texttt{recover}(h)$ represents a state in which a transient fault has occurred and has been caught. The labels in history h were visited during the execution. The state $\texttt{hwerror}(h)$ represents a state in which a transient fault causes transition to an invalid address. Figure 2 summarizes the syntax of the assembly language and machine states.

3.1 Dynamic Semantics

We model the dynamic semantics of the assembly language using a small step operational semantics. In general, the single step operational judgments have the form $\Sigma \longrightarrow_k \mathcal{F}$ where k, which is either zero or one, records the number of faults that occur during the step.

The Fault Model. The most interesting rules in the system are the rules modeling faults. The primary rule (*zap-reg*) simply states that the value in any register may be corrupted arbitrarily, though its color tag (which has no operational significance) remains unchanged.

$$\frac{R(r) = c\ n}{(C, h, R, b) \longrightarrow_1 (C, h, R[r \mapsto c\ n'], b}\ (\textit{zap-reg})$$

The rule above may fire at any time. In particular, it may fire just prior to execution of a jump ($\texttt{jmp}\ r_t$) or a branch ($\texttt{brz}\ r_z\ r_t$), corrupting the jump target in register r_t. Such a fault models a control-flow error. Of course, it is equally possible that any other register is corrupted.

For uniformity in our fault model, we also consider errors in execution of the $\texttt{recovernz}\ r_z$ instruction. Recall, this instruction is merely a macro for the conditional branch $\texttt{brz}\ r_z\ \ell_{recover}$. However, since $\ell_{recover}$ is a constant, it is unaffected by faults in registers modeled by the *zap-reg* rule (our other branching instructions take arguments in registers). To simulate a fault that causes control to jump somewhere other than the $\ell_{recover}$ label when the r_z register contains a non-zero value, we add the following rules.

$$\frac{R_{val}(r_z) \neq 0}{(C, h, R, \texttt{recovernz}\ r_z; b) \longrightarrow_1 (C, h, R, C(\ell))}\ (\textit{zap-recovernz1})$$

$$\frac{R_{val}(r_z) \neq 0}{(C, h, R, \texttt{recovernz}\ r_z; b) \longrightarrow_1 \texttt{hwerror}(h)}\ (\textit{zap-recovernz2})$$

The *zap-recovernz1* rule expresses the possibility that a fault causes execution to jump to some random block labeled ℓ rather than the recovery code block. The *zap-recovernz2* rule expresses the possibility that

$$\frac{}{(C, h, R, \texttt{movi}\ r_d\ v; b) \longrightarrow_0 (C, h, R[r_d \mapsto v], b)}\ (\textit{movi})$$

$$\frac{v' = R_{col}(r_a)\ (R_{val}(r_a) - R_{val}(r_b))}{(C, h, R, \texttt{sub}\ r_d\ r_a\ r_b; b) \longrightarrow_0 (C, h, R[r_d \mapsto v'], b)}\ (\textit{sub})$$

$$\frac{}{(C, h, R, \texttt{intend}\ r_t; b) \longrightarrow_0 (C, h, R[r_i \mapsto R(r_t)], b)}\ (\textit{intend})$$

$$\frac{R_{val}(r_z) = 0}{(C, h, R, \texttt{intendz}\ r_z\ r_t; b) \longrightarrow_0 (C, h, R[r_i \mapsto R(r_t)], b)}\ (\textit{intendz-set})$$

$$\frac{R_{val}(r_z) \neq 0}{(C, h, R, \texttt{intendz}\ r_z\ r_t; b) \longrightarrow_0 (C, h, R, b)}\ (\textit{intendz-unset})$$

$$\frac{R_{val}(r_z) = 0}{(C, h, R, \texttt{recovernz}\ r_z; b) \longrightarrow_0 (C, h, R, b)}\ (\textit{recovernz-ok})$$

$$\frac{R_{val}(r_z) \neq 0}{(C, h, R, \texttt{recovernz}\ r_z; b) \longrightarrow_0 \texttt{recover}(h)}\ (\textit{recovernz-halt})$$

$$\frac{R_{val}(r_z) = 0 \qquad R_{val}(r_t) \in Dom(C)}{(C, h, R, \texttt{brz}\ r_z\ r_t) \longrightarrow_0 (C, (h, R_{val}(r_t)), R[r_i \mapsto O\ R_{val}(r_i)], C(R_{val}(r_t)))}\ (\textit{brz-taken})$$

$$\frac{R_{val}(r_z) \neq 0 \qquad \ell{+}1 \in Dom(C)}{(C, h, R, \texttt{brz}\ r_z\ r_t) \longrightarrow_0 (C, (h, \ell{+}1), R[r_i \mapsto O\ R_{val}(r_i)], C(\ell{+}1))}\ (\textit{brz-untaken})$$

$$\frac{R_{val}(r_z) = 0 \qquad R_{val}(r_t) \notin Dom(C)}{(C, h, R, \texttt{brz}\ r_z\ r_t) \longrightarrow_0 \texttt{hwerror}(h)}\ (\textit{brz-hw-error})$$

$$\frac{R_{val}(r_t) \in Dom(C)}{(C, h, R, \texttt{jmp}\ r_t) \longrightarrow_0 (C, (h, R_{val}(r_t)), R[r_i \mapsto O\ R_{val}(r_i)], C(R_{val}(r_t)))}\ (\textit{jmp})$$

$$\frac{R_{val}(r_t) \notin Dom(C)}{(C, h, R, \texttt{jmp}\ r_t) \longrightarrow_0 \texttt{hwerror}(h)}\ (\textit{jmp-hw-error})$$

Figure 3: Operational Semantics.

Static Expressions

exp kinds	κ	::=	$\kappa_{int} \mid \kappa_{hist}$
exp contexts	Δ	::=	$\cdot \mid \Delta, x : \kappa$
exps	e	::=	$x \mid n \mid e - e \mid e?e : e$
substitutions	S	::=	$\cdot \mid S, e/x$

Types

stage description	ρ	::=	$check \mid ok \mid go \mid goz$
basic types	τ	::=	$int \mid \rho \mid \forall[\Delta](\Gamma, \sigma)$
value types	t	::=	$\langle c, \tau, e \rangle$
type option	$\tau\ opt$	::=	$\tau \mid$ undef

Context Typing

heap typing	Ψ	::=	$\cdot \mid \Psi, \ell \to \tau$
reg file types	Γ	::=	$\cdot \mid \Gamma, r \to t$
history typing	σ	::=	$\epsilon \mid x \mid \sigma \circ e$
zap tags	Z	::=	$\cdot \mid c \mid CF$

Figure 4: Typing Syntax.

a fault causes control to jump to an illegal address. Attempted execution of code at this address results in immediate transition to the final state `hwerror`(h), where h represents the sequence of blocks visited not including the illegal address.

Other Operational Rules. All other operational rules are presented in Figure 3. The majority of these rules are quite unsurprising. For instance, the *movi* rule implements the move by updating the register file. Notice that the index on the arrow is "0" indicating no fault occurs during this transition. Naturally, the *intend* rule is very similar to *movi* as `intend` is just a macro for a move into r_i.

Skipping to the bottom of the figure, it is important to notice there are two rules for expressing the semantics of a `jmp` r_t instruction. The *jmp* rule fires whenever r_t contains the address of a valid block. Of course, due to a fault earlier in execution, the address in r_t may not be the intended destination for this jump. In addition to transferring control to the new block, this instruction does some bookkeeping. In particular, it extends the current history with the destination address and it changes the color of r_i to be orange. The latter effect facilitates the proof of correctness and will be explained in further detail in Section 4. The second rule for `jmp` semantics fires whenever r_t does *not* contain the address of a valid block. In this case, there is an attempt to transfer control to an illegal address, which is caught by the hardware. The rules for conditional branches follow a similar pattern to those for the unconditional jumps.

4 Typing

The design of the type system is based on three main concepts:

- Classifying the reliability properties of values.
- Using abstract types to make sure that the fault tolerance protocol proceeds in the correct order, with no steps omitted or inappropriate steps inserted.
- Equivalence checking to ensure that redundant values act as proper backups to the original.

The following paragraphs explain the main intuitions behind each concept. Later subsections will give precise details.

Classifying the Reliability Properties of Values. Since faults occur completely unpredictably and at run time, it is not possible for the type system to know which values have incurred faults or to track the propagation of presumed faulty values precisely. It is not possible to know exactly values may or may not be trusted. Consequently, as is usual, the type system will have to approximate these properties somehow. It does so by assigning each value to one of several compile-time "groups" and ensuring that each member of a group has related reliability properties. As a mnemonic, each group has an associated color c, which may be either *green*, *blue* or *orange*.

As we saw in Section 2, the protocol for detecting faults in software involves keeping redundant copies of the values used in control flow transfers and using these to check for correct control flow. We will refer to the main computation as the *green* computation, and the redundant copies as the *blue* (or "backup") computation. Most values either belong to the green group or to the blue group. These two groups have the property that they are *redundant* and *independent*. In other words, a fault in a green value can never percolate to a blue value and vice versa. Consequently, when corresponding green and blue values are compared at least one of them must be correct, even when a fault has occurred. This mutual independence property is ensured by a series of simple checks in the type system that guarantee that green values are not used to construct blue values and vice versa.

But what if a control-flow fault *has* occurred? In that case, almost all program invariants are invalidated, including any properties of either blue or green values. Fortunately, though, the defining characteristic of *orange* values is preservation of their properties in just this situation.

There are two general mechanisms by which one can guarantee orange values maintain their expected properties in the face of a control-flow fault. The first mechanism is to ensure that the orange value in question is not live across the control-flow transfer: If the value has been constructed in the current block and does not depend upon values in previous blocks, a control-flow error will not influence its properties. This first mechanism is used in the checking code at the beginning of each program block. In particular, the operation that moves a label into a register at the beginning of a block may label its results orange:

```
Lk: movi r2, Lk          // r2 is orange
    ...
    sub r2, r2, ri
    ...
    recovernz r2
    ...
```

The second mechanism involves ensuring that every possible control-flow transfer maintains the invariant in question. If the invariant is true across *every* control-flow transfer, then it is true no matter where control winds up. This second mechanism is used to classify the contents of r_i as orange across every control-flow transfer. Just as the type system isolates green values from blue and blue from green, orange is also isolated from the other two. Again, the purpose is to avoid having a fault in one color influence the others.

While values are classified using colors, entire machine states are classified using a related concept called *zap tags*. Intuitively, each zap tag specifies which colors may no longer be trusted. For example, if zap tag Z is empty (written "$\cdot$"), then there have been no faults during the computation, and all values, no matter what their color, satisfy the standard invariants associated with their compile-time type. On the other hand, if Z is a color c, then there has been a fault to a value colored c and, moreover, the corruption may have spread to any other value colored c. Consequently, values colored c will not necessarily satisfy any particular properties associated with their compile-time type.

The final zap tag CF classifies machine states after a control-flow error has occurred. In this case, control may have transfered somewhere totally unexpected, and so we know nothing about green *or* blue values. Fortunately, though, the properties of orange values remain valid.

Figure 5 summarizes the properties that hold under each zap tag while in block ℓ. We say a value is *trusted* if it satisfies standard canonical forms properties (*e.g.,* a value with code type is actually a pointer to valid code). We say a value is *untrusted* when we cannot guarantee standard canonical forms properties hold.

Zap Tag	G **values**	B **values**	O **values**	ℓ **correct**
$\cdot$	trusted	trusted	trusted	yes
G	*untrusted*	trusted	trusted	yes
B	trusted	*untrusted*	trusted	yes
O	trusted	trusted	*untrusted*	yes
CF	*untrusted*	*untrusted*	trusted	*no*

Figure 5: Properties of colored values and zap tags.

We say a zap tag Z is a subtype of another Z', written $Z \leq Z'$, when the values in machine states classified by Z are more trusted than the values in machine states classified by Z'. Hence the empty zap tag is a subtype of all other zap tags, and both B and G zap tags are a subtype of CF.

Typing Protocol Stages. The instructions in each block can be thought of as being divided into three distinct stages – the *checking code*, the *block body*, and the *exit code*. Each of these stages has its own distinct invariants. The type of intentions register r_i encodes the current stage and ensures that the stages occur in the correct order. It also guarantees no part of the protocol can be omitted or any inappropriate instruction added. These stages may be summarized as follows.

1. The checking code compares the intended target with the current location to determine if there has been a control flow fault. In this region, r_i must be colored orange and have basic type *check*.
2. In the block body, we already know the control flow correctly transferred to this block. At the end of this sequence, there is some green register that holds the target label for the next control flow transfer and some blue register that holds the duplicate copy of this label. In the absence of faults, these two values are equal. In this region, r_i must have basic type *ok*.
3. The exit code sequence sets the intended target and transfers control to the new block. In the exit code sequence, r_i is colored blue and has type *go* when an intention has been set, and type *goz* when a conditional intention has been set. As we saw in Section 3.1, r_i is recolored orange during the execution of the control flow transfer.

For example, consider the example code sequences from Section 2 shown in Figure 6. On entry, each block first checks that control has reached this block correctly, and sets its intentions before transferring control to another block.

Testing Value Equivalence. There are many places in the fault tolerance protocol where we require a blue value to be an independent and redundant copy of a green value. To ensure that blue and green values are equal in the absence of faults, we characterize them accurately using a language of static expressions. Moreover, many of the typing rules require that corresponding expressions are equal.

Onward. Now that we have summarized the intuitions behind the main concepts, we will proceed with the technical details.

4.1 Value Typing

The type of a value is a triple $\langle c, \tau, e \rangle$. The color c is assigned according to the intuitions expressed in the previous subsection. A basic type τ is either an integer, a code type, or a special type that indicates the state of the fault tolerance protocol.

The third component e is a static expression that describes the value in more detail. These expressions are used to require that blue and green computations compute identical results in the absence of faults.

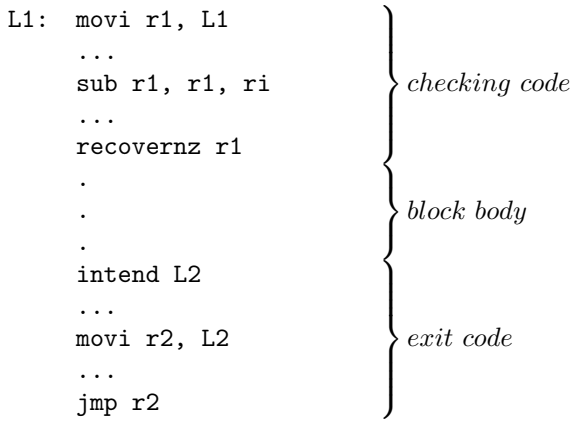

Figure 6: Example: Protocol Stages.

These expressions include variables x, integers n, subtraction $e_1 - e_2$ and conditional expressions $e_1?e_2 : e_3$ which equal e_2 when e_1 is non-zero and e_3 when e_1 is zero.

Expression judgments are shown in Figure 7. The kinding judgment $\Delta \vdash e : \kappa$ holds when all the free variables in e are contained in Δ. Expression e has kind κ_{int} when it describes an integer and kind κ_{hist} when it describes a history typing. Expression variables x are the only expressions that can have type κ_{hist}. Judgments $\Delta \vdash \sigma$ wf, $\Delta \vdash \Gamma$ wf, and $\Delta \vdash \tau$ wf hold when all expressions used in these constructs are well-kinded. The judgment $\Delta \vdash S : \Delta'$ holds when S provides substitutions for all variables in Δ', and the substituted expressions are well-formed in Δ.

The function $[\![e]\!]$ supplies the denotation of the closed static expression e as an integer. The judgments $\Delta \vdash e_1 = e_2$ and $\Delta \vdash e_1 \neq e_2$ hold when the relation holds for all substitutions of the variables in Δ. $\Delta \vdash \sigma_1 = \sigma_2$ simply extends this relationship to all expressions in the two sequences.

Value Typing Judgment. The value typing judgment has the form $\Delta; \Psi \vdash^Z v : t$ and is shown in Figure 8. The context Δ contains free expression variables, and the heap type Ψ maps integer addresses to basic types. The zap tag Z characterizes the current state of the machine as explained earlier. Z is always the empty tag when a user checks a program at compile time. It only takes on other values at run time for the purposes of the proof of preservation.

The main value typing judgment depends upon an auxiliary judgment with the form $\Psi \vdash n : \tau$. This auxiliary judgment allows integer n to be given either a basic *int* type, a stage description type ρ, or a code type $\Psi(n)$. If e is equal to n and $\Psi \vdash n : \tau$, then $c\ n$ can always be given the type $\langle c, \tau, e \rangle$. However, if the zap tag Z is a color c, then all values $c\ n$ can also be typed using any basic type and any well-formed expression — such a general rule reflects the fact that we can make no guarantees about such values. When the zap tag is CF, then *any* green and blue value can be given *any* type, including giving green values blue types and vice versa. In other words, as mentioned earlier, when there has been a control-flow fault, all bets are off for green and blue values.

Value Subtyping. There is also a subtyping relationship $\Delta \vdash t \leq t'$. This judgment allows type $\langle c, \tau, e \rangle$ to be subtype of $\langle c, int, e' \rangle$ whenever $\Delta \vdash e = e'$. Register File subtyping is a basic extension of value subtyping that requires every register in the first register file type to be a subtype of the corresponding register in the second. The subtyping judgment is used to type check control flow transfers.

$\boxed{\Delta \vdash e : \kappa}$

$$\frac{x \in Dom(\Delta)}{\Delta \vdash x : \Delta(x)}\ (wf\text{-}var) \qquad \frac{}{\Delta \vdash n : \kappa_{int}}\ (wf\text{-}int)$$

$$\frac{\Delta \vdash e_1 : \kappa_{int} \qquad \Delta \vdash\ e_2 : \kappa_{int}}{\Delta \vdash e_1 - e_2 : \kappa_{int}}\ (wf\text{-}sub) \qquad \frac{\Delta \vdash e_1 : \kappa_{int} \qquad \Delta \vdash e_2 : \kappa \qquad \Delta \vdash e_3 : \kappa}{\Delta \vdash e_1\ ?\ e_2 : e_3\ : \kappa}\ (wf\text{-}ifexp)$$

$\boxed{\Delta \vdash \sigma \text{ wf}}$

$$\frac{}{\Delta \vdash \epsilon \text{ wf}}\ (wf\text{-}\sigma\text{-}\epsilon) \qquad \frac{\Delta \vdash e : \kappa}{\Delta \vdash \sigma \circ e \text{ wf}}\ (wf\text{-}\sigma)$$

$\boxed{\Delta \vdash \Gamma \text{ wf}}$

$$\frac{\forall r.\ \ \Gamma(r) = \langle c, \tau, e\rangle \ \ \wedge \ \ \Delta \vdash e : \kappa}{\Delta \vdash \Gamma \text{ wf}}\ (wf\text{-}R)$$

$\boxed{\Delta \vdash \tau \text{ wf}}$

$$\frac{}{\Delta \vdash int \text{ wf}}\ (wf\text{-}int) \qquad \frac{}{\Delta \vdash \rho \text{ wf}}\ (wf\text{-}\rho) \qquad \frac{(\Delta \cup \Delta') \vdash \Gamma' \text{ wf} \qquad (\Delta \cup \Delta') \vdash \sigma' \text{ wf}}{\Delta \vdash \forall[\Delta'](\Gamma', \sigma') \text{ wf}}\ (wf\text{-}\forall[\Delta'](\Gamma', \sigma'))$$

$\boxed{\Delta \vdash S : \Delta'}$

$$\frac{}{\Delta \vdash \cdot : \cdot}\ (subst\text{-}emp\text{-}t) \qquad \frac{\Delta \vdash S : \Delta' \qquad \Delta \vdash e : \kappa \qquad x \notin (\Delta \cup \Delta')}{\Delta \vdash S, e/x : (\Delta', x : \kappa)}\ (subst\text{-}t)$$

$\boxed{[[e]]}$

$$\begin{array}{lcl} [\![n]\!] & = & n \\ [\![e_1 - e_2]\!] & = & [\![e_1]\!] - [\![e_2]\!] \\ [\![e_b\ ?\ e_t : e_f]\!] & = & \ if\ [\![e_b]\!]\ then\ [\![e_t]\!]\ else\ [\![e_f]\!] \end{array}$$

$\boxed{\Delta \vdash e = e}$

$$\frac{\Delta \vdash e_1 : \kappa_{int} \qquad \Delta \vdash e_2 : \kappa_{int} \qquad \forall S.\ \ \cdot \vdash S : \Delta \Longrightarrow [[S(e_1)]] = [[S(e_2)]]}{\Delta \vdash e_1 = e_2}\ (e\text{-}eq)$$

$$\frac{\Delta \vdash e_1 : \kappa_{int} \qquad \Delta \vdash e_2 : \kappa_{int} \qquad \forall S.\ \ \cdot \vdash S : \Delta \Longrightarrow [[S(e_1)]] \neq [[S(e_2)]]}{\Delta \vdash e_1 \neq e_2}\ (e\text{-}neq)$$

$\boxed{\Delta \vdash \sigma = \sigma}$

$$\frac{}{\Delta \vdash \epsilon = \epsilon}\ (\epsilon\text{-}eq) \qquad \frac{\Delta \vdash e_1 = e_2 \qquad \Delta \vdash \sigma_1 = \sigma_2}{\Delta \vdash \sigma_1 \circ e_1 = \sigma_2 \circ e_2}\ (\sigma\text{-}eq)$$

Figure 7: Static Expression Judgments.

$\boxed{\Psi \vdash n : \tau}$

$$\frac{}{\Psi \vdash n : int}\ (int\text{-}t) \quad \frac{}{\Psi \vdash n : \Psi(n)}\ (address\text{-}t) \quad \frac{}{\Psi \vdash n : \rho}\ (\rho\text{-}t)$$

$\boxed{\Delta; \Psi \vdash^{Z} v : t}$

$$\frac{\Psi \vdash n : \tau \qquad \Delta \vdash e = n}{\Delta; \Psi \vdash^{Z}\ c\ n :\ \langle c, \tau, e\rangle}\ (val\text{-}t)$$

$$\frac{\Delta \vdash e : \kappa_{int}}{\Delta; \Psi \vdash^{c}\ c\ n :\ \langle c, \tau, e\rangle}\ (val\text{-}zap\text{-}c\text{-}t) \qquad \frac{\Delta \vdash e : \kappa_{int} \qquad c' = B\ or\ c' = G}{\Delta; \Psi \vdash^{CF}\ c\ n :\ \langle c', \tau, e\rangle}\ (val\text{-}zap\text{-}CF\text{-}t)$$

$\boxed{\Delta \vdash t \leq t'}$

$$\frac{\Delta \vdash e_1 = e_2}{\Delta \vdash\ \langle c, \tau, e_1\rangle\ \leq\ \langle c, \tau, e_2\rangle}\ (subtp\text{-}reflex) \qquad \frac{\Delta \vdash e_1 = e_2}{\Delta \vdash\ \langle c, \tau, e_1\rangle\ \leq\ \langle c, int, e_2\rangle}\ (subtp\text{-}int)$$

$\boxed{\Delta \vdash \Gamma \leq \Gamma'}$

$$\frac{\forall r.\ \Gamma_1(r) \leq \Gamma_2(r)}{\Delta \vdash \Gamma_1 \leq \Gamma_2}\ (\Gamma\text{-}subtp)$$

Figure 8: Value Typing Judgment and Subtyping Judgment.

$\boxed{\Delta; \Psi; \Gamma \vdash i : \Gamma'}$

$$\frac{r_d \neq r_i}{\Delta; \Psi; \Gamma \vdash \texttt{movi}\ r_d\ c\ n : \Gamma[r_d \mapsto \langle c, int, n\rangle]}\ (movi\text{-}t)$$

$$\frac{r_d \neq r_i \qquad \Gamma(r_a) = \langle c, int, e_a\rangle \qquad \Gamma(r_b) = \langle c, int, e_b\rangle}{\Delta; \Psi; \Gamma \vdash \texttt{sub}\ r_d\ r_a\ r_b : \Gamma[r_d \mapsto \langle c, int, e_a - e_b\rangle]}\ (sub\text{-}t)$$

$$\frac{\Gamma(r_i) = \langle c_i, ok, e_i\rangle \qquad \Gamma(r_t) = \langle B, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle}{\Delta; \Psi; \Gamma \vdash \texttt{intend}\ r_t : \Gamma[r_i \mapsto \langle B, go, e_t\rangle]}\ (intend\text{-}t)$$

$$\frac{\begin{array}{l}\Gamma(r_i) = \langle B, go, e_i\rangle \qquad \Gamma(r_t) = \langle B, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle \\ \Gamma(r_z) = \langle B, int, e_z\rangle \qquad t' = \langle B, goz, e_z ? e_i : e_t\rangle\end{array}}{\Delta; \Psi; \Gamma; \vdash \texttt{intendz}\ r_z\ r_t : \Gamma[r_i \mapsto t']}\ (intendz\text{-}t)$$

Figure 9: Instruction Typing Judgment.

4.2 Instruction Typing

Figure 9 presents the instruction typing judgment, which has the form $\Delta; \Psi; \Gamma \vdash i : \Gamma'$. As before, Δ contains free expression variables and Ψ types heap addresses. Γ acts as the precondition for the instruction, mapping registers to their corresponding types prior to execution of the instruction. Γ' acts as the postcondition for the instruction, mapping registers to types guaranteed after execution of the instruction.

The simplest instruction to type check is the `movi` r_d c n instruction. It merely updates the type of r_d to be $\langle c, int, n\rangle$. The subtraction instruction `sub` r_d r_a r_b requires that the values being subtracted are integers. Notice it also requires the integers arguments have the same color as the result – this restriction prevents faults in values with one color to influence another. These two instructions place no restrictions on the type of r_i, so they can occur during any stage of a block.

The unconditional intention instruction `intend` r_t requires that r_i has basic type *ok*. This restriction guarantees any new intend will occur after the checking code has been completed. Intentions are part of the blue computation, so the register that is used to set the intention must contain a blue value with code type. The type of r_i is updated to reflect the new static expression and the new stage *go*.

The conditional intention instruction `intendz` r_z r_t is similar, although it must occur after an unconditional intention. In other words, to set intentions for a conditional branch, first use `intend` to set r_i to contain the address of the fall through block, and then conditionally set it to contain the branch target. The resulting type of r_i has basic type *goz* and a conditional expression guarded by the expression e_z describing r_z. If e_z is nonzero, then r_i will be described by e_i, which describes the fall through branch. Otherwise, it is described by e_t, which describes the branch target.

Despite the fact that `recovernz` is syntactically an instruction, it is type-checked using the block typing judgment because it affects the set of free expression variables.

4.3 Block Typing

The block typing judgment $\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b$ contains a number of new pieces of information. In addition to Δ, Ψ, and Γ, the block typing judgment is parameterized by a sequence σ, an expression e_i, and a type option $\tau\ opt$.

The sequence σ contains a list of expressions that describe the locations in the current history h. While typing a block at location ℓ, σ has the form $x_h \circ \ell$ meaning that the program has already visited some unknown sequence of locations (x_h) leading up to this point and that the label of the current block is ℓ. The judgment $\Delta \vdash \sigma_1 = \sigma_2$ holds when each expression in σ_1 is equal to the corresponding expression in σ_2 for all substitutions of the variables in Δ.

The expression e_i describes the intended target when the transfer occurred to the current label ℓ. If control flow correctly transfered to ℓ, then $\Delta \vdash e_i = \ell$.

The option type $\tau\ opt$ contains the type of the label $\ell + 1$ if such a label exists. It is used when a branch falls through to the subsequent block to determine the type of that block.

The block typing rules are presented in Figure 10. The first rule, *sequence-t*, is used when the first instruction in a block is one of the basic instructions described previously. Descriptions of the other rules follow.

Recovery. There are three distinct rules for checking `recovernz` r_z. All of them require the instruction to occur in the first stage of the block when r_i contains an orange value with basic type *check*. The operand register r_z compares this value to the current label.

The first rule *recovernz-t* applies when r_i is described by variable x_i. This is the rule used by a programmer to check correctness of their program at compile time. Control only proceeds past this point in the block if x_i is equal to the expression e_ℓ, which describes the current location, so the remainder of the block is typed by substituting e_ℓ for x_i. The types of r_i and r_z are updated to reflect the deletion of x_i. Judgment $\Delta \vdash \Gamma/r_i/r_z$ wf and $\Delta \vdash \sigma$ wf hold when all variables used in registers other than r_i and r_z as well as the expressions in σ are all contained in Δ. Since none of these pieces of state contain x_i, they do not need to be modified.

$\boxed{\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b}$

$$\frac{\Delta; \Psi; \Gamma \vdash i : \Gamma' \qquad \Delta; \Psi; \Gamma'; \sigma; e_i; \tau\ opt \vdash b}{\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash i; b}\ (\textit{sequence-t})$$

$$\frac{\begin{array}{l}\Gamma(r_i) = \langle O, check, x_i\rangle \\ \Gamma(r_z) = \langle O, int, e_z\rangle \\ \Delta, x : \kappa_{int} \vdash e_z = e_\ell - x_i \\ \Delta \vdash \Gamma/r_i/r_z\ \text{wf} \qquad \Delta \vdash \sigma\ \text{wf} \qquad \Delta \vdash e_\ell : \kappa_{int} \\ \Gamma' = \Gamma[r_z \mapsto\ \langle O, int, 0\rangle][r_i \mapsto\ \langle B, ok, e_\ell\rangle] \\ \Delta; \Psi; \Gamma'; \sigma \circ e_\ell; e_\ell; \tau\ opt \vdash b\end{array}}{(\Delta, x : \kappa_{int}); \Psi; \Gamma; \sigma \circ e_\ell; x_i; \tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_z) =\ \langle O, int, e_z\rangle & \cdot \vdash e_z = e_i - e_\ell \\ \Gamma(r_i) = \langle O, check, e_i\rangle & \cdot \vdash e_i = e_\ell \\ \multicolumn{2}{l}{\cdot; \Psi; \Gamma[r_i \mapsto \langle O, ok, e_i\rangle]; \sigma \circ e_\ell; e_i; \tau\ opt \vdash b}\end{array}}{.; \Psi; \Gamma; \sigma \circ e_\ell; e_i; \tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-eq-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_z) = \langle O, int, e_z\rangle & \cdot \vdash e_z = e_i - e_\ell \\ \Gamma(r_i) = \langle O, check, e_i\rangle & \cdot \vdash e_i \neq e_\ell\end{array}}{.; \Psi; \Gamma; \sigma \circ e_\ell; e_i; \tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-neq-t})$$

$$\frac{\begin{array}{l}\Gamma(r_i) = \langle B, goz, e'_z ? e'_f : e'_t\rangle \\ \Delta \vdash e'_f = e_\ell + 1 \\ \Gamma(r_z) = \langle G, int, e_z\rangle \\ \Delta \vdash e_z = e'_z \\ \Gamma(r_t) = \langle G, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle \\ \Delta \vdash e_t = e'_t \\ \exists S_t\ .\ \Delta \vdash S_t : \Delta_t \\ \Delta \vdash \Gamma[r_i \mapsto\ \langle O, check, e'_t\rangle] \leq S_t(\Gamma_t) \\ \Delta \vdash \sigma \circ e_\ell \circ e'_t = S_t(\sigma_t) \\ \exists S_f\ .\ \Delta \vdash S_f : \Delta_f \\ \Delta \vdash \Gamma[r_i \mapsto\ \langle O, check, e'_f\rangle] \leq S_f(\Gamma_f) \\ \Delta \vdash \sigma \circ e_\ell \circ e'_f = S_f(\sigma_f)\end{array}}{\Delta; \Psi; \Gamma; \sigma \circ e_\ell; e_i; \forall[\Delta_f](\Gamma_f, \sigma_f) \vdash \texttt{brz}\ r_z\ r_t}\ (\textit{brz-t})$$

$$\frac{\begin{array}{l}\Gamma(r_i) = \langle B, go, e'_t\rangle \\ \Gamma(r_t) = \langle G, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle \\ \Delta \vdash e_t = e'_t \\ \exists S_t\ .\ \Delta \vdash S_t : \Delta_t \\ \Delta \vdash \Gamma[r_i \mapsto \langle O, check, e'_t\rangle] \leq S_t(\Gamma_t) \\ \Delta \vdash \sigma \circ e_\ell \circ e_t = S_t(\sigma_t)\end{array}}{\Delta; \Psi; \Gamma; \sigma \circ e_\ell; e_i; t \vdash \texttt{jmp}\ r_t}\ (\textit{jmp-t})$$

Figure 10: Block Typing Judgment.

The other two rules *recovernz-eq-t* and *recovernz-neq-t* are needed to carry out the proof of type preservation (particularly the substitution lemma), but would never be used to type check programs prior to execution. In these situations, x_i has already been replaced with a closed expression e_i that describes the intentions register at block entry. Here, it is evident that either $\cdot \vdash e_i = e_\ell$ or not, so there is one typing rule for each situation. The rule *recovernz-neq-t* does not place any requirements on the remainder of the block since control does not proceed past this point.

Control Flow Transfers. In order to verify unexpected transfers from the end of one block to the beginning of any other, code blocks must have the same basic precondition. To be specific, each block must expect that the intentions register r_i contains an orange value with basic type *check* that is described by a variable x_i. This variable does not occur anywhere else in the function precondition. This condition entails every target block can accept any orange value in r_i.

The rule *jmp-t* requires that r_i has type $\langle B, go, e'_t \rangle$ specifying that the intention must already have been set before the jump. Also, the current jump target has a code type and is described by an expression e_t that is equal to e'_t. This enforces that in the absence of faults, the duplicate target is equal to the target.

The target label precondition contains a set of expression variables Δ_t and requires a register file described by Γ_t and a history described by σ_t. There is some substitution S_t for the variables in Δ_t so that the current register file type and sequence are subtypes of those required by the target. (The register file subtyping judgment is a straightforward extension of the value subtyping judgment.)

The `jmp` r_t and `brz` r_z r_t instructions recolor the blue intention register to be orange when control is transfered to a new block. At first, this seems to contradict the rule that faults to a value of one color should never corrupt values of other colors. However, because the target block doesn't place any restrictions on the expression describing r_i, the variable x_i that describes the value can be instantiated with the value itself. Because of this, a blue value that is not trusted can become a trusted orange value during a control flow transfer, continuing to leave only the blue values untrusted.

The rule *brz-t* is similar, but adds in the conditional register r_z and specifies both the fall through and the branch cases.

4.4 Machine State Typing

Code Memory Typing. The judgment $\vdash C : \Psi$ describes the invariants for code memory. As described previously, all blocks must have the same basic precondition. The register r_i is described by the type $\langle O, check, x_i \rangle$. The other registers are colored either blue or green, and their static expressions do not contain the variable x_i. If a label ℓ has type $\forall[\Delta](\Gamma, x_h \circ \ell)$, then code at that label must be well-typed given Ψ, Δ, Γ, $x_h \circ \ell$, the intention expression x_i, and the fall through label type $\Psi(\ell + 1)$.

Register File Typing. The judgment $\Psi \vdash^Z R : \Gamma$ states that register file R has type Γ under zap tag Z given heap typing Ψ. It holds when each register in R has the corresponding type in Γ under Z. And again, values with colors that are affected by Z are not trusted to have their given types.

History Typing. A history h is described by sequence σ when each location is equal to the corresponding expression.

Machine State Typing. A machine state Σ is well-typed under zap tag Z when each of its elements is well-typed, and two additional invariants hold. (1) If Z is *CF* then the current location ℓ is not equal to the intended location e_i. Otherwise, if Z is not *CF*, then these two are equal. (2) If the current block b has proceeded past the checking stage, then it must be the case that ℓ is equal to e_i. These two invariants together imply it is not possible for code past the checking stage of a block to be well-typed under the *CF* zap tag. Consequently, a proof of type preservation will imply that any control-flow error will be caught in the checking stage of the next block.

$\boxed{\vdash C : \Psi}$

$$\frac{\begin{array}{l}\forall \ell \in Dom(C) \cup Dom(\Psi)\ . \\ \quad \Psi(\ell) = \forall[\Delta](\Gamma, x_h \circ \ell) \\ \quad \Delta = \Delta',\ x_i : \kappa_{int},\ x_h : \kappa_{hist} \\ \quad \Gamma = \Gamma',\ r_i \mapsto \langle O, check, x_i \rangle \\ \quad \forall r' \in Dom(\Gamma')\ .\ \ \Gamma'(r') \neq \langle O, \tau', e' \rangle \\ \quad \Delta' \vdash \Psi(\ell+1)\ \text{wf} \qquad \Delta' \vdash \Gamma'\ \text{wf} \\ \quad \Delta; \Psi; \Gamma; x_h \circ \ell; x_i; \Psi(\ell+1) \vdash C(\ell)\end{array}}{\vdash C : \Psi}\ (C\text{-}t)$$

$\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r.\ \cdot; \Psi \vdash^Z R(r) : \Gamma(r)}{\Psi \vdash^Z R : \Gamma}\ (R\text{-}t)$$

$\boxed{\vdash h : \sigma}$

$$\frac{}{\vdash () : \epsilon}\ (h\text{-}empty\text{-}t) \qquad \frac{\vdash h : \sigma \qquad \cdot \vdash e = n}{\vdash (h, n) : \sigma \circ e}\ (h\text{-}app\text{-}t)$$

$\boxed{\vdash^Z (C, h, R, b)}$

$$\frac{\begin{array}{l}\vdash C : \Psi \\ \Psi \vdash^Z R : \Gamma \\ \vdash (h, \ell) : \sigma \\ (Z = CF)\ ?\ \ (\cdot \vdash e_i \neq \ell)\ \ :\ \ (\cdot \vdash e_i = \ell) \\ \Gamma(r_i) \neq \langle O, check, e_i \rangle \implies .\vdash e_i = \ell \\ \cdot; \Psi; \Gamma; \sigma; e_i; \Psi(\ell+1) \vdash b\end{array}}{\vdash^Z (C, (h, \ell), R, b)}\ (\Sigma\text{-}t)$$

Figure 11: Machine State Typing.

4.5 Typing Lemmas

In this section, we briefly explain the main lemmas used to prove type safety.

Expression Equality. Expression equality is transitive. Conditional expression $e_z?e_f : e_t$ is equal to either e_t or e_f depending on the value of e_z.

Lemma 1 (Expression Equality)

1. *If* $\Delta \vdash e_1 = e_2$ *and* $\Delta \vdash e_2 = e_3$ *then* $\Delta \vdash e_1 = e_3$
2. *If* $\Delta \vdash \sigma_1 = \sigma_2$ *and* $\Delta \vdash \sigma_2 = \sigma_3$ *then* $\Delta \vdash \sigma_1 = \sigma_3$
3. *If* $\Delta \vdash e_z = 0$ *then* $\Delta \vdash e_z?e_f : e_t = e_t$.
4. *If* $\Delta \vdash e_z \neq 0$ *then* $\Delta \vdash e_z?e_f : e_t = e_f$.

Proof: *By the definition of* $\Delta \vdash e = e$ *and* $\Delta \vdash \sigma = \sigma$ *and the definition of* $[\![e]\!]$.

Substitution. Substituting an expression of kind κ for a free variable of kind κ preserves typing. Applying a substitution S that provides substitutions for a number of free variables also preserves typing.

Lemma 2 (Substitution)

1. *If* $\Delta, x : \kappa \vdash e' : \kappa'$ *and* $\Delta \vdash e : \kappa$ *then* $\Delta \vdash e'[e/x] : \kappa'$
2. *If* $\Delta, x : \kappa \vdash e_1 = e_2$ *and* $\Delta \vdash e : \kappa$ *then* $\Delta \vdash e_1[e/x] = e_2[e/x]$
3. *If* $(\Delta, x : \kappa); \Psi \vdash^Z v : t$ *and* $\Delta \vdash e : \kappa$ *then* $\Delta; \Psi \vdash^Z v : t[e/x]$
4. *If* $(\Delta, x : \kappa); \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b$ *and* $\Delta \vdash e : \kappa$ *then* $\Delta; \Psi; \Gamma[e/x]; \sigma[e/x]; e_i[e/x]; \tau\ opt[e/x] \vdash b$
5. *If* $(\Delta_1, \Delta_2) \vdash e' : \kappa'$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1 \vdash S(e') : \kappa'$
6. *If* $(\Delta_1, \Delta_2) \vdash e_1 = e_2$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1 \vdash S(e_1) = S(e_2)$
7. *If* $(\Delta_1, \Delta_2); \Psi \vdash^Z v : t$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1; \Psi \vdash^Z v : S(t)$
8. *If* $(\Delta_1, \Delta_2); \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1; \Psi; S(\Gamma); S(\sigma); S(e_i); S(\tau\ opt) \vdash b$

Proof:

1. *By induction on the structure of* $\Delta, x : \kappa \vdash e' : \kappa'$
2. *By case analysis on the structure of* $\Delta, x : \kappa \vdash e_1 = e_2$ *using Part 1.*
3. *By case analysis on the structure of* $(\Delta, x : \kappa); \Psi \vdash^Z v : t$ *using Parts 1 and 2.*
4. *By induction on the structure of* $(\Delta, x : \kappa); \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b$ *using Parts 1-3. The case for rule (recovernz-t) divides into two subcases depending on if* $e_z = 0$ *and uses rule (recovernz-eq-t) or rule (recovernz-neq-t) as appropriate.*

5-8. *By induction on the size of* Δ_2*, using Parts 1-4 respectively.*

Subtyping. If a value has a type t and this type is a subtype of t', then the value can also be given type t'.

Lemma 3 (Subtyping)
If $\Delta \vdash t \leq t'$ *and* $\Delta; \Psi \vdash^Z v : t$ *then* $\Delta; \Psi \vdash^Z v : t'$

Proof: *By induction on the derivation of* $\Delta; \Psi \vdash^Z v : t$. *Each case uses inversion of the subtyping rules and Lemma 1 (Expression Equality).*

Canonical Forms. If a value $c\ n$ has type $\langle c', \tau', e' \rangle$ under zap tag Z, then our knowledge about n depends both on the base type τ' and also the relationship between Z and c'.

If $Z = CF$ and the color in the type is G or B, then the judgment may be been derived using rule (*val-zap-CF-t*), so we know nothing about n. However, we do know that the expression has kind κ_{int}.

If Z is c', then the judgment may have been derived by rule (*val-zap-c-t*), so again we only know that the expression e' has kind κ_{int}.

The remaining case applies when Z is not equal to the color in the type c' and when either $Z \neq CF$ or the color in the type is neither G nor B. In this case the judgment must have been derived using rule (*val-t*), so we know the color tag on the value is equal to the color tag in the type and the expression e' is equal to the value n. In addition, if τ' is a code type, we also know that n is a valid code address.

Lemma 4 (Canonical Forms)
If $\cdot\ ; \Psi \vdash^Z c\ n : \langle c', \tau', e' \rangle$ *and* $\vdash C : \Psi$ *then*

1. *If* $Z = CF$ *and* ($c' = B$ *or* $c' = G$), *then* $\cdot \vdash e' : \kappa_{int}$.
2. *If* $Z = c'$ *then* $c = c'$ *and* $\cdot \vdash e' : \kappa_{int}$.
3. *If* $Z \neq c'$ *and* ($Z \neq CF$ *or* ($c' \neq B$ *and* $c' \neq G$)) *then*
   - $c = c'$
   - $\cdot \vdash e' = n$
   - $\tau' = \forall[\Delta](\Gamma, \sigma) \implies n \in Dom(C)$

Proof: *By inspection of* $\cdot; \Psi \vdash^Z c\ n : \langle c', \tau', e' \rangle$.

Color Weakening. If a value has a type t under zap tag Z, then that value also has type t under any zap tag Z' that is a supertype of Z.

Lemma 5 (Color Weakening)
If $\Delta; \Psi \vdash^Z v : t$ *and* $Z \leq Z'$ *then* $\Delta; \Psi \vdash^{Z'} v : t$

Proof: *By case analysis of* $\Delta; \Psi \vdash^Z v : t$ *and the definition of* $Z \leq Z'$.

4.6 Type Safety

We have proven that the TAL_{CF} type system is sound using the standard notion of Progress and Preservation. Progress asserts that machine states well-typed under the empty zap tag can take a step to another ordinary machine state. States that are well-typed under any zap can also take a step, but this step may reach any state, including `recover`(h) or `hwerror`(h).

Theorem 6 (Progress)
1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow_0 \Sigma'$.

2. *If* $\vdash^Z \Sigma$ *then* $\Sigma \longrightarrow_0 \mathcal{F}$.

Proof: *The proof for each part is by case analysis on the current block b of* Σ *using Lemma 1 (Expression Equality) and Lemma 4 (Canonical Forms). The complete proof appears in the online appendix [14].*

Preservation states that execution preserves typing. States well-typed under the empty zap tag continue to be so after taking a non-faulty step. States typed under any zap also remain well-typed after a non-faulty step, but the zap tag may escalate to a supertype. This elevation might occur at control flow transfers. A zap tag of B or G becomes CF whenever the corruption has spread to the operands being used in the transfer. This way the block that results from the transfer can be well-typed under CF even when control has transferred to a totally unexpected block. The intentions register is always the only orange value that is live across control flow transfers, and we have already seen that it is well-typed even when a control fault has occurred. Finally, a state is well-typed under the empty zap tag and takes a faulty step, then the resulting state is well-typed under some color c.

Theorem 7 (Preservation)

1. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow_0 \Sigma'$ *then* $\vdash \Sigma'$
2. *If* $\vdash^Z \Sigma$ *and* $\Sigma \longrightarrow_0 \Sigma'$ *then* $\exists Z' \,.\, \vdash^{Z'} \Sigma'$ *and* $Z \leq Z'$.
3. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow_1 \Sigma'$ *then* $\exists c \,.\, \vdash^c \Sigma'$

Proof: *The proof for each part is by case analysis on the corresponding single step judgment using Lemma 1 (Expression Equality), Lemma 4 (Canonical Forms), and Lemma 5 (Color Weakening). Cases for the jump and branch rules also use Lemma 2 (Substitution) and Lemma 3 (Subtyping). The complete proof appears in the online appendix [14].*

5 Fault Tolerance Theorem

In this section, we first present a handful of definitions and lemmas relating machine states to other states, and then use these to formally state and prove the Fault Tolerance Theorem. Brief proof sketches are provided, and complete proofs are available in the online appendix [14].

5.1 Machine State Simulation

We say that a faulty value simulates a fault-free value under color c if the values are equal when they are not colored by c.

$$\frac{}{c'\ n \overset{c}{\sim} c'\ n}\ (\textit{sim-val}) \qquad \frac{}{c\ n \overset{c}{\sim} c\ n'}\ (\textit{sim-val-zap})$$

A faulty machine state Σ_f simulates a fault-free state Σ under color c if Σ_f is well-typed under c, Σ is well-typed under the empty zap tag, and the two states are identical modulo the values in registers colored c.

$$\frac{\vdash (C,h,R,b) \qquad \vdash^c (C,h,R_f,b) \qquad \forall r.R_f(r) \overset{c}{\sim} R(r)}{(C,h,R_f,b) \overset{c}{\sim} (C,h,R,b)}\ (\textit{sim-}\Sigma)$$

5.2 Block Execution

The Block Step Lemma states that given a non-faulty computation and a corresponding faulty version Σ_f, if the non-faulty computation can take a non-faulty step to some other state *in the same block*, then the faulty computation will either also take a step within the current block or will take a single step to the recover state.

Lemma 8 (Block Step)
If $\Sigma_f \overset{c}{\sim} (C, h, R, b)$ *and* $(C, h, R, b) \longrightarrow_0 (C, h, R', b')$ *then either*

1. $\Sigma_f \longrightarrow_0 \Sigma'_f$ *and* $\Sigma'_f \overset{c}{\sim} (C, h, R', b')$, *or*
2. $\Sigma_f \longrightarrow_0 \texttt{recover}(h)$

Proof: *By case analysis of* $(C, h, R, b) \longrightarrow_0 (C, h, R', b')$ *and Theorem 7 (Preservation).*

In order to reason about block execution, we extend the single step relation $\Sigma \longrightarrow_k \Sigma'$ from Section 3 to create the judgment $\Sigma \rightsquigarrow_k \mathcal{F}$ which states that $\mathcal{F}$ is the result of executing the *current block* of Σ while incurring k faulty transitions. Execution proceeds up to the control-flow transfer statement at the end of the current block or the recover state if the block terminates prematurely by transitioning to recovery code. For example, if $\Sigma = (C, h, R, i_1; ...; i_n; \texttt{jmp}\ r_t)$, then either $\mathcal{F} = (C, h, R', \texttt{recover}(h))$ or $\mathcal{F} = (C, h, R', \texttt{jmp}\ r_t)$.

$\boxed{\Sigma \rightsquigarrow_k \mathcal{F}}$

$$\frac{(C, h, R, b) \longrightarrow_0 \texttt{recover}(h)}{(C, h, R, b) \rightsquigarrow_0 \texttt{recover}(h)}\ (\textit{blk-eval-recover})$$

$$\frac{}{(C, h, R, \texttt{jmp}\ r_t) \rightsquigarrow_0 (C, h, R, \texttt{jmp}\ r_t)}\ (\textit{blk-eval-jmp})$$

$$\frac{}{(C, h, R, \texttt{brz}\ r_z\ r_t) \rightsquigarrow_0 (C, h, R, \texttt{brz}\ r_z\ r_t)}\ (\textit{blk-eval-brz})$$

$$\frac{(C, h, R, b) \longrightarrow_{k_1} (C, h, R', b') \qquad (C, h, R', b') \rightsquigarrow_{k_2} \mathcal{F}}{(C, h, R, b) \rightsquigarrow_{(k_1+k_2)} \mathcal{F}}\ (\textit{blk-eval-sequence})$$

The Block Execution Lemma states that given a faulty computation Σ_f that simulates a non-faulty computation, the result of executing the faulty block will either simulate the result of executing the non-faulty block, or executing the faulty block will result in $\texttt{recover}(h)$.

Lemma 9 (Block Execution)
If $\Sigma_f \overset{c}{\sim} (C, h, R, b)$ *and* $(C, h, R, b) \rightsquigarrow_0 \Sigma'$ *then either*

1. $\Sigma_f \rightsquigarrow_0 \Sigma'_f$ *and* $\Sigma'_f \overset{c}{\sim} \Sigma'$, *or*
2. $\Sigma_f \rightsquigarrow_0 \texttt{recover}(h)$

Proof: *By induction on the structure of* $(C, h, R, b) \rightsquigarrow_0 \Sigma'$ *and Lemma 8 (Block Step).*

5.3 Fault Recovery

The *CF* Fault Step Lemma states that once a control flow fault has occurred, execution will either step within the same block or will step to recovery code.

Lemma 10 (*CF* Fault Step)
If $\vdash^{CF} (C, h, R, b)$ *then either*

1. $(C, h, R, b) \longrightarrow_0 (C, h, R', b')$ *and* $\vdash^{CF} (C, h, R', b')$
2. $(C, h, R, b) \longrightarrow_0 \texttt{recover}(h)$.

Proof: *By case analysis on the structure of* b *using Theorem 7 (Preservation).*

The Fault Recovery Lemma states that once a control flow fault has occurred, control will always reach recovery code before exiting the current block.

Lemma 11 (*CF* Fault Block Execution)
If $\vdash^{CF} (C, h, R, b)$ *then* $(C, h, R, b) \rightsquigarrow_0 \texttt{recover}(h)$.

Proof: *By induction on the length of* b *and Lemma 10 (CF Fault Step).*

5.4 Block Transitions

In order to reason about transitions *between* blocks, we define the judgment $\Sigma \Longrightarrow^{\ell} \Sigma'$ whenever $(C, h, R, b) \longrightarrow_0 (C, (h, \ell), R', b')$. In other words, control transfers from the end of one block to the beginning of another block ℓ in a single step.

$\boxed{\Sigma \Longrightarrow^{\ell} \Sigma'}$

$$\frac{(C, h, R, b) \longrightarrow_0 (C, (h, \ell), R', b')}{(C, h, R, b) \Longrightarrow^{\ell} (C, (h, \ell), R', b')} \ (\textit{trans-eval})$$

The Block Transition Lemma states that whenever a non-faulty computation transitions to a new block, the corresponding faulty computation will either (1) transition to the same block and continue to be indistinguishable from the non-faulty computation, (2) trigger a hardware error, or (3) transition to an incorrect block where the error will be detected before control leaves the incorrect block.

Lemma 12 (Block Transition)
If $\Sigma_f \overset{c}{\sim} (C, h, R, b)$ *and* $(C, h, R, b) \Longrightarrow^{\ell} \Sigma'$ *then either*

1. $\Sigma_f \Longrightarrow^{\ell} \Sigma'_f$ *and* $\Sigma'_f \overset{c}{\sim} \Sigma'$
2. $\Sigma_f \longrightarrow_0 \texttt{hwerror}(h)$
3. $\Sigma_f \Longrightarrow^{\ell'} \Sigma'_f$ *and* $\Sigma'_f \rightsquigarrow_0 \texttt{recover}(h, \ell')$

Proof: *By case analysis of the structure of* $(C, h, R, b) \Longrightarrow^{\ell} \Sigma'$ *and Lemma 11 (CF Fault Block Execution).*

5.5 Program Execution

The judgment $\Sigma \implies_k^h \mathcal{F}$ states that machine state Σ executes through a sequence of blocks h to reach state $\mathcal{F}$ while incurring k faulty transitions. In other words, if $\Sigma = (C, h_1, R, b)$, then $\mathcal{F}$ is either $(C, (h_1, h), R', \texttt{jmp}\ r_t)$, $(C, (h_1, h), R', \texttt{brz}\ r_z\ r_t)$, $\texttt{hwerror}(h_1, h)$, or $\texttt{recover}(h_1, h)$.

$\boxed{\Sigma \implies_k^h \Sigma'}$

$$\frac{\Sigma \rightsquigarrow_k \mathcal{F}}{\Sigma \implies_k^{()} \mathcal{F}}\ (\textit{prog-exec-blk})$$

$$\frac{\Sigma \implies_k^h \Sigma' \qquad \Sigma' \longrightarrow_0 \texttt{hwerror}(h', h)}{\Sigma \implies_k^h \texttt{hwerror}(h', h)}\ (\textit{prog-exec-seq-hwerror})$$

$$\frac{\Sigma \implies_{k_1}^h \Sigma' \qquad \Sigma' \implies^{\ell} \Sigma'' \qquad \Sigma'' \rightsquigarrow_{k_2} \mathcal{F}}{\Sigma \implies_{(k_1+k_2)}^{(h,\ell)} \mathcal{F}}\ (\textit{prog-exec-seq-trans-blk})$$

The Faulty Execution Lemma states that if a faulty execution Σ_f simulates a non-faulty execution Σ under some color c, then Σ_f behaves in one of four possible ways with regards to Σ. (1) Executing Σ_f results in the same sequence of blocks h as executing Σ and the resulting faulty state simulates the corresponding non-faulty state under the same color c. (2) Executing Σ_f results in an attempt to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the execution of Σ_f visits the same blocks as the execution of Σ. (3) While executing Σ_f, a fault is detected and control is transferred to recovery code even though no incorrect blocks have been visited. This situation can be caused by a fault affecting the intentions register or the checking code. (4) While executing Σ_f, control veers off course to a block that is not visited in the execution of Σ. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Lemma 13 (Faulty Execution)
If $\Sigma_f \overset{c}{\sim} \Sigma$ and $\Sigma \implies_0^h \Sigma'$ then either:

1. $\Sigma_f \implies_0^h \Sigma'_f$ *and* $\Sigma'_f \overset{c}{\sim} \Sigma'$
2. $\Sigma_f \implies_0^{h_f} \texttt{hwerror}(h', h_f)$ *and h_f is a prefix of h*
3. $\Sigma_f \implies_0^{h_f} \texttt{recover}(h', h_f)$ *and h_f is a prefix of h*
4. $\Sigma_f \implies_0^{h_f} \texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

Proof: *By induction on the structure of $\Sigma \implies_0^h \Sigma'$, Lemma 9 (Block Execution) and Lemma 12 (Block Transition).*

5.6 The Fault Tolerance Theorem

A program is fault-tolerant if any execution of the program with a single fault behaves in one of four possible ways with regards to the original, non-faulty computation. (1) The faulty computation visits the same sequence of blocks as the original and the resulting faulty state simulates the corresponding original state under some color c. (2) The faulty computation attempts to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the faulty computation visits the same blocks as the original computation. (3) The faulty computation detects a fault in software and jumps to recovery code even though no incorrect blocks have been visited. This situation can be caused by a fault affecting the intentions register or the checking code. (4) The faulty computation veers off course to a block that does not match the corresponding block in the original computation. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Theorem 14 (Fault Tolerance)
If $\vdash \Sigma$ *and* $\Sigma \Longrightarrow_0^h \Sigma'$ *then at least one of the following cases applies and all derivations* $\Sigma \Longrightarrow_1^{h_f} \mathcal{F}$ *where* $length(h_f) \leq length(h)$ *fit one of these cases:*

1. $\Sigma \Longrightarrow_1^h \Sigma'_f$ *and* $\exists c \,.\, \Sigma'_f \overset{c}{\sim} \Sigma'$

2. $\Sigma \Longrightarrow_1^{h_f} \texttt{hwerror}(h', h_f)$ *and* h_f *is a prefix of* h

3. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* h_f *is a prefix of* h

4. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

Proof: *By case analysis on the structure of* $\Sigma \Longrightarrow_0^h \Sigma'$. *In essence, arbitrarily divide the computation* $\Sigma \Longrightarrow_0^h \Sigma'$ *into two pieces with some* Σ'' *as the intermediate state. Use one of the fault rules to step* Σ'' *to* Σ''_f. *If* c *is the color of the value that faults, then* $\Sigma''_f \overset{c}{\sim} \Sigma''$. *Then use Lemma 13 (Faulty Execution) with* $\Sigma''_f \overset{c}{\sim} \Sigma''$ *and the remainder of the computation from* Σ'' *to* Σ' *to determine what happens after the fault. Use these results and the first half of the computation to show that one of the four cases applies to the entire computation containing a single fault.*

6 Translation

In order to show that TAL_{CF} is sufficiently expressive to be of interest, we define a simple language of while loops, and show how to compile statements in this language into well-typed TAL_{CF} programs.

6.1 A Simple While Loop Language

The while loop language statements consist of simple assignment, subtraction, if statements, while loops, and sequences of statements. As all the variables in this language contain integers, the well-formedness judgment $V \vdash s$ simply enforces that all variables v in s exist in the variable context V.

$$\begin{array}{rcl} s & ::= & v := n \mid v_d := v_a - v_b \\ & & \mid \texttt{if0}\ v_z\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ v_z \neq 0\ \texttt{do}\ s \\ & & \mid s_1; s_2 \end{array}$$

6.2 Checking Code and Exit Code Macros

The translation rules and lemmas make use of the following macros that implement the protocol from Section 2. These macros make use of two temporary registers: t_g and t_b.

Macro "check ℓ" generates the checking code at the entry of block ℓ to check that control has correctly transferred to this block. Macro "intendjmp ℓ_t" sets the intention and then executes the jump to target block ℓ_t. Finally, macro "intendzbrz r_z ℓ_t ℓ_f" uses register r'_z to conditionally set the intention to fall through to block ℓ_f or branch to block ℓ_t, and then uses register r_z to execute the conditional branch.

$$\begin{array}{rcl}
\text{check } \ell & \equiv & \texttt{movi } t_g\ O\ \ell; \texttt{sub } t_g\ t_g\ r_i; \\
& & \texttt{recovernz } t_g \\
\\
\text{intendjmp } \ell_t & \equiv & \texttt{movi } t_b\ B\ \ell_t; \texttt{intend } t_b; \\
& & \texttt{movi } t_g\ G\ \ell_t; \texttt{jmp } t_g \\
\\
\text{intendzbrz } r_z\ \ell_t\ \ell_f & \equiv & \texttt{movi } t_b\ B\ \ell_f; \texttt{intend } t_b; \\
& & \texttt{movi } t_b\ B\ \ell_t; \texttt{intendz } r'_z\ t_b; \\
& & \texttt{movi } t_g\ G\ \ell_t; \texttt{brz } r_z\ t_g
\end{array}$$

6.3 Translating Variable Context V

Since all variables are considered live at all program points, every assembly-level instruction block will have essentially the same signature. If the context V contains variable v_1,...,v_n, then each block in the translation requires $2n+3$ registers: a green copy r_k and a blue copy r'_k for each variable v_k, the intention register r_i, and two temporary registers t_g and t_b. (We have made no effort to optimize this translation, it merely serves to demonstrate the theoretical expressiveness of the target language.)

The function $[\![V]\!]_\ell$ generates the code type of the code at label ℓ. The generated Δ contains all the expression variables that are need in Γ and σ. Each label will have a slightly different history typing σ, since the sequence ends with the current label. Register file typing Γ gives types to each of the $2n+3$ registers. Register r_k is green and r'_k is blue. Both registers have basic type *int* and are described by the same expression variable x_k, which enforces that they are equal on entry to the block. Register r_i is orange, has basic type *check*, and is described by expression variable x_i. Again, since we have not optimized the translation, we will assume that during block transitions t_g always contains a green value and t_b always contains a green value. They may hold values of other colors during the body of a block.

$\boxed{[\![V]\!]_\ell = \forall[\Delta](\Gamma,\sigma)}$

$$\frac{\begin{array}{l}
\texttt{choose fresh variables } x_1,\ ...,\ x_n,\ x_h,\ x_{r_i},\ x_g,\ x_b,\ x_o \\
\Delta = x_1 : \kappa_{int},\ \ \ldots,\ x_n : \kappa_{int},\ x_h : \kappa_{hist},\ x_{r_i} : \kappa_{int},\ x_g : \kappa_{int},\ x_b : \kappa_{int} \\
\sigma = x_h \circ \ell \\
\Gamma = \{\ \ r_1 : \langle G, int, x_1\rangle,\ r'_1 : \langle B, int, x_1\rangle, \ldots,\ r_n : \langle G, int, x_n\rangle,\ r'_n : \langle B, int, x_n\rangle, \\
\qquad r_i : \langle O, check, x_{r_i}\rangle, \\
\qquad t_g : \langle G, int, x_g\rangle,\ t_b : \langle B, int, x_b\rangle\ \ \}
\end{array}}{[\![v_1, \ldots, v_n]\!]_\ell = \forall[\Delta](\Gamma,\sigma)}\ (\textit{trans-V})$$

The function $\texttt{Gen}\Psi(V, L)$ computes the heap typing Ψ that maps each label in L to its corresponding type

$\boxed{\texttt{Gen}\Psi(V, L)}$

$$\begin{array}{l}
\texttt{Gen}\Psi(V, \cdot) = \cdot \\
\texttt{Gen}\Psi(V, (L, \ell)) = \texttt{Gen}\Psi(V, L),\ \ell \mapsto [\![V]\!]_\ell
\end{array}$$

6.4 Partial Translations

A 4-tuple of objects $(L, C, \vec{i}, \ell)$ is used to track the code generated during the translation. C is the code memory that contains all blocks generated so far. L contains labels that may be referred to by blocks in C but whose corresponding blocks have not yet been generated. ℓ is the label that will be assigned to the block that is currently being generated. $\vec{i}$ contains the list of instructions for this block that have been generated so far. The instructions for checking the checking code and exit code are not included and will be added when the block is added to C.

The judgment $V; \Psi_1 \vdash C : \Psi_2$ is used to type code memory C as it is being generated. There are two disjoint heaping typings: Ψ_1 contains labels that may be referenced by C but whose corresponding code blocks may not have been generated yet, and Ψ_2 contains the types for the blocks that have already been generated. Both Ψ_1 and Ψ_2 map each label ℓ to $[\![V]\!]_\ell$. In addition, each label in Ψ_2 has a type that can be used to type check corresponding block.

$\boxed{V; \Psi_1 \vdash C : \Psi_2}$

$$\frac{\begin{array}{l} Dom(\Psi_1) \cap Dom(\Psi_2) = \emptyset \\ \forall \ell \in Dom(\Psi_1)\ .\ \Psi_1(\ell) = [\![V]\!]_\ell \\ Dom(C) = Dom(\Psi_2) \\ \forall \ell \in Dom(\Psi_2)\ . \\ \quad \Psi_2(\ell) = [\![V]\!]_\ell = \forall[\Delta](\ (\Gamma, r_i \mapsto \langle O, check, x_i \rangle),\ x_h \circ \ell\ ) \\ \quad \wedge\ \ \Delta;\ (\Psi_1 \cup \Psi_2);\ (\Gamma, r_i \mapsto \langle O, check, x_i \rangle);\ (x_h \circ \ell);\ x_i;\ (\Psi_1 \cup \Psi_2)(\ell+1)\ \ \vdash\ \ C(\ell) \end{array}}{V; \Psi_1 \vdash C : \Psi_2}\ (C\text{-}wf)$$

Judgment $V \vdash \vec{i}$ wf states that $\vec{i}$ is a sequence of pairs of instructions that perform duplicate moves and subtractions. For example, the following is a well-formed list of instructions.

$$\texttt{movi}\ r_3\ G\ 3;\ \texttt{movi}\ r'_3\ B\ 3;\ \texttt{sub}\ r_4\ r_5\ r_6;\ \texttt{sub}\ r'_4\ r'_5\ r'_6; \ldots$$

Using these definitions, we say a partial translation $(L, C, \vec{i}, \ell)$ is well-formed when the code memory C is well-formed using the heap typings calculated from the label ℓ and the labels in L and the labels already in the domain of C. In addition, the instruction list $\vec{i}$ is well-formed.

$\boxed{V \vdash (L, C, \vec{i}, \ell)\ \text{wf}}$

$$\frac{\begin{array}{l} \Psi_1 = \texttt{Gen}\Psi(V, (L, \ell)) \\ \Psi_2 = \texttt{Gen}\Psi(V, Dom(C)) \\ V; \Psi_1 \vdash C : \Psi_2 \\ V \vdash \vec{i}\ \text{wf} \end{array}}{V \vdash (L, C, \vec{i}, \ell)\ \text{wf}}\ (\textit{partial-trans-}\ \text{wf})$$

The Block Construction Lemma says that the instruction list $\vec{i}$ from a well-formed partial translation can be used to construct a block by adding checking code to the beginning and exit code to the end. The exit code can refer to any existing label ℓ' as the jump target. The exit code can be a conditional branch only if the fall-through block $\ell + 1$ exists. The new code memory formed by adding this new block is also well-formed.

Lemma 15 (Block Construction)
If $V \vdash (L, C, \vec{i}, \ell)$ wf *then* $\forall \ell' \in ((L, \ell) \cup Dom(C))$.

1. $\texttt{Gen}\Psi(V, L)\ \vdash\ C[\ \ell \mapsto \text{check}\ \ell;\ \vec{i};\ \text{intendjmp}\ \ell'\ ] : \texttt{Gen}\Psi(V, (Dom(C), \ell))$
2. *If* $l + 1 \in ((L, \ell) \cup Dom(C))$
   then $\texttt{Gen}\Psi(V, L)\ \vdash\ C[\ \ell \mapsto \text{check}\ \ell;\ \vec{i};\ \text{intendzbrz}\ r_z\ \ell'\ \ell + 1\ ] : \texttt{Gen}\Psi(V, (Dom(C), \ell))$

Proof: *Using the macro definitions, the definition of* $V \vdash (L, C, \vec{i}, \ell)$ wf*, and instruction typing rules from Section 4.2.*

6.5 Translating Statements

The main translation judgment $[\![V \vdash s]\!](L, C, \vec{i}, \ell) = (L', C', \vec{i}', \ell')$ extends the existing partial translation $(L, C, \vec{i}, \ell)$ with the translation of statement s.

The statement translation rules are shown in Figure 12. Translating simple assignment and subtraction statements simply adds pairs of assembly instructions to the end of the current instruction sequence. Sequencing two statements uses the partial translation from the first statement to translate the second.

Translating `if0` statements requires the addition of new blocks: ℓ_f contains the fallthrough branch, ℓ_t contains the true branch, and ℓ_m is where the two branches merge. The function $NumBlock(s)$ calculates the number of blocks generated by the translation of s. The current block b_ℓ contains checking code, the code $\vec{i}$ generated for the block so far, and ends with a conditional branch to ℓ_t (and an automatic fallthrough to ℓ_f). The new label ℓ_t is the starting point for the code generated for the true branch s_1. The ending label of this code ℓ'_t finishes by merging back to the common block at ℓ_m. The translation of the false branch is similar. The final code memory contains all blocks generated by either branch as well as the blocks ending each branch by jumping to the merge block ℓ_m. The label in the resulting partial translation is ℓ_m.

Translating `while` statements also requires the addition of new blocks. The current block at ℓ is terminated with an unconditional jump to a beginning block at ℓ_b that tests the condition and branches to an ending label ℓ_e if the condition fails. Otherwise it falls through to the block at ℓ_s which contains the translation of s and terminates with a jump back to the beginning block. The label in the resulting partial translation is ℓ_e.

The Statement Translation Lemma says that given a well-formed partial translation $(L, C, \vec{i}, \ell)$, translating a statement s results in another well-formed partial translation. In addition, the new set of undefined labels L' is equal to that in the original partial translation.

Lemma 16 (Statement Translation)
If $[\![V \vdash s]\!](L, C, \vec{i}, \ell) = (L', C', \vec{i}', \ell')$ *and* $V \vdash (L, C, \vec{i}, \ell)$ wf
then $V \vdash (L', C', \vec{i}', \ell')$ wf *and* $L = L'$

Proof: *Using the definition of* $V \vdash (L, C, \vec{i}, \ell)$ wf *and Lemma 15 (Block Construction).*

6.6 The Translation Theorem

To translate a statement s as a stand-alone program, it is translated as in the previous section with 1 as the starting label. Because there is no halt instruction in TAL_{CF}, code is added to the last block in the translation to create an infinite loop at label ℓ_{halt}. The function InitRegFile(V) creates an initial register file that maps each register used to translate V to 0.

The assembly language program corresponding to s is the TAL_{CF} state consisting of the generated code memory, a history with only the first location, an initial register file, and code to jump to the first label in code memory. If the original statement is well-formed, then the translation is well-typed.

Theorem 17 (Translation)
If $[\![V \vdash s]\!](., ., ., 1) = (., C, \vec{i}, \ell)$ *then*
$\vdash$ ($C[\ell \mapsto \text{check } \ell; \vec{i}; \text{intendjmp } \ell_{halt}][\ell_{halt} \mapsto \text{check } \ell_{halt}; \text{intendjmp } \ell_{halt}]$, 0, InitRegFile($V$), intendjmp 1)

Proof: *Using Lemma 16 (Statement Translation), Lemma 15 (Block Construction), the block typing rules from Section 4.3, and the machine state typing rules from Section 4.4.*

$\boxed{[\![V \vdash s]\!](L, C, \vec{i}, \ell) = (L', C', \vec{i}', \ell')}$

$$\frac{\vec{i}' = \vec{i};\ \texttt{movi}\ r_k\ G\ n;\ \texttt{movi}\ r'_k\ B\ n}{[\![V \vdash v := n]\!](L, C, \vec{i}, \ell) = (L, C, \vec{i}', \ell)}\ (t\text{-}assign)$$

$$\frac{\vec{i}' = \vec{i};\ \texttt{sub}\ r_k\ r_a\ r_b;\ \texttt{sub}\ r'_k\ r'_a\ r'_b}{[\![V \vdash v_d := v_a - v_b]\!](L, C, \vec{i}, \ell) = (L, C, \vec{i}', \ell)}\ (t\text{-}sub)$$

$$\frac{\begin{array}{l}[\![V \vdash s_1]\!](L, C, \vec{i}, \ell) = (L_1, C_1, \vec{i}_1, \ell_1) \\ [\![V \vdash s_2]\!](L_1, C_1, \vec{i}_1, \ell_1) = (L_2, C_2, \vec{i}_2, \ell_2)\end{array}}{[\![V \vdash s_1; s_2]\!](L, C, \vec{i}, \ell) = (L_2, C_2, \vec{i}_2, \ell_2)}\ (t\text{-}seq)$$

$$\frac{\begin{array}{l}\ell_f = \ell + 1 \\ \ell_t = \ell_f + NumBlocks(s_1) \\ \ell_m = \ell_t + NumBlocks(s_2) \\ \\ b_\ell = \text{check } \ell;\ \vec{i};\ \text{intendzbrz } r_z\ \ell_f\ \ell_t \\ \\ [\![V \vdash s_1]\!]((L, \ell_f), C[\ell \mapsto b_\ell], \cdot, \ell_t) = (L'_t, C'_t, \vec{i}'_t, \ell'_t) \\ b'_t = \text{check } \ell'_t;\ \vec{i}'_t;\ \text{intendjmp } \ell_m \\ \\ [\![V \vdash s_2]\!]((L, \ell_t), C[\ell \mapsto b_\ell], \cdot, \ell_f) = (L'_f, C'_f, \vec{i}'_f, \ell'_f) \\ b'_f = \text{check } \ell'_f;\ \vec{i}'_f;\ \text{intendjmp } \ell_m \\ \\ C' = (C'_t \cup C'_f)[\ell'_t \mapsto b'_t][\ell'_f \mapsto b'_f]\end{array}}{[\![V \vdash \texttt{if0}\ v_z\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!](L, C, \vec{i}, \ell) = (L, C', \cdot, \ell_m)}\ (t\text{-}if)$$

$$\frac{\begin{array}{l}\ell_b = \ell + 1 \\ \ell_s = \ell_b + 1 \\ \ell_e = \ell_s + NumBlocks(s) \\ \\ C'' = \quad C[\ell \mapsto \text{check } \ell;\ \vec{i};\ \text{intendjmp } \ell_b] \\ \qquad\qquad [\ell_b \mapsto \text{check } \ell_b;\ \text{intendzbrz } r_z\ \ell_e\ \ell_s] \\ \\ [\![V \vdash s]\!]((L, \ell_e), C'', \cdot, \ell_s) = (L'_s, C'_s, \vec{i}'_s, \ell'_s) \\ \\ C' = C'_s[\ell'_s \mapsto \text{check } \ell'_s;\ \vec{i}'_s;\ \text{intendjmp } \ell_b]\end{array}}{[\![V \vdash \texttt{while}\ v_z \neq 0\ \texttt{do}\ s]\!](L, C, \vec{i}, \ell) = (L'_s, C', \cdot, \ell_e)}\ (t\text{-}while)$$

Figure 12: Translation of While Programs

7 Related Work

There is a long history of research into techniques for delivering fault tolerance in the presence of transient faults. What sets the current work apart from the vast majority of the literature is the use of a provably sound type system to verify reliability properties of low-level code. As mentioned in the introduction, this research follows previous work on λ_{zap} [21] and TAL_{FT} [13]. However, neither λ_{zap} nor TAL_{FT} provided software mechanisms for guaranteeing control-flow integrity. Recently, Elsman [6] has shown how to extend λ_{zap} so that the atomic voting operations can be broken down into a series of conditional statements. However, again, there is no treatment of control-flow.

Perhaps the most closely related work to the current paper is CFI, a provably-sound technique for enforcing control-flow integrity in a security context [1, 2]. The goal of CFI is to guarantee that machine code obeys a predefined "control-flow policy" that constrains the sequence of blocks control can move through. The key distinction between CFI and our own work is the threat model, which makes all the difference. CFI attackers can modify arbitrary amounts of machine state in arbitrary ways; this sort of attacker models the threat posed by buffer-overflow vulnerabilities effectively. However, CFI attackers cannot touch three reserved registers during the execution of certain code sequences. Protecting against transient faults is, on the one hand, easier, because the attacker can only modify a single value as opposed to arbitrary amounts of state arbitrarily many times, but, on the other hand, more difficult, because no single bit of state can be a priori guaranteed to be protected. On balance, it appears that having just that tiny bit of protected state makes the solution and proof of correctness of the CFI problem simpler than the corresponding fault tolerance problem. For instance, the CFI checker can be defined as a relatively straightforward series of context-insensitive conditions on the code; there is no need for a sophisticated type system. It is also the case that the structure of the desired theorems are somewhat different. In the case of CFI, the running code must satisfy a security policy specified as a control-flow graph. In our case, the desired end result is a simulation theorem that guarantees that every faulty run of the program is properly related to the non-faulty run.

Our work builds upon many past research efforts in fault tolerance, particularly those that deal with control-flow checking. For example, Oh *et al.* [11] developed a pure software control-flow checking scheme (CFCSS) where each control transfer generates a run-time signature that is validated at each block entry point. The SWIFT system [16], another software-only fault tolerance system, also uses signature checking very much like that in the current paper. Venkatasubramanian, Hayes and Murray [20] proposed a technique called Assertions for Control Flow Checking (ACFC) that assigns an execution parity to each basic block and detects faults based on parity errors. Schuette and Shen [18] explored control-flow monitoring (ARC) to detect transient faults affecting the program flow on a Multiflow TRACE 12/300 machine with little extra overhead. Ohlsson and Rimen [12] developed a technique to monitor software control flow signatures without building a control flow graph. However, this latter technique requires additional hardware: A coprocessor is used to dynamically compute the signature from the running instruction stream and a watchdog timer is used to detect the absence of block signatures. The distinguishing feature of our research is not the control-flow checking procedure itself, but the type system we designed to verify the code and our proof that well-typed programs are indeed fault tolerant. These previous efforts did not rigorously specify the properties they intended to enforce nor did they prove their techniques actually enforce them.

Naturally, our research also builds upon previous work in the verification of low-level code including the original typed assembly language (TAL) [8] and proof-carrying code (PCC) [9]. However, both TAL and PCC operate under the assumption of nonfaulty hardware and therefore ignore the major issues of reliability on which this paper has focused.

8 Conclusion

Current trends in hardware design including increased transistor density, decreased voltages and increased clock rates are decreasing the reliability of modern processors. While these effects are currently limited, for the most part, to high-end clusters and supercomputing facilities, they pose a broader threat to future

systems. One way to counter this trend is to shift some of the burden for reliability into software. However, reasoning about the correctness of software running on faulty hardware is an extremely difficult task, particularly when faults may affect program control flow.

In this paper, we defined a simple abstract machine that exhibits control-flow faults and we analyzed the correctness of a software protocol for detecting them. Our analysis proceeded through the definition of a type system that guarantees programs are reliable relative to a simple fault model. From a theoretical perspective, the type system serves as a tool for reasoning about the correctness of faulty programs. From a practical perspective, it may be implemented and used as a debugging tool in compilers that purport to generate reliable code. We have rigorously proven strong reliability properties for our type system and have shown it is sufficiently expressive to serve as the target for compilation of a simple language of while programs. Overall, we believe this is the first successful attempt at reasoning rigorously about software mechanisms for controlling control flow faults.

Acknowledgments

We would like to thank Andrew Appel, David August, George Reis and Neil Vachharajani for many enlightening discussions on transient faults, hardware mechanisms and fault tolerance.

This research is funded in part by NSF award CNS-0627650 and a Microsoft fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Microsoft.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, Nov. 2005.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, Nov. 2005.

[3] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[4] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[5] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In *IEEE Micro*, volume 25, pages 10–16, December 2005.

[6] M. Elsman. Fault-tolerant voting in a simply-typed lambda calculus. Technical Report ITU-TR-2007-99, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, June 2007.

[7] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[9] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Langauges*, pages 106–119, Paris, Jan. 1997.

[10] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[12] J. Ohlsson and M. Rimen. Implicit signature checking. In *International Conference on Fault-Tolerant Computing*, June 1995.

[13] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *International Symposium on Programming Language Design and Implementation (PLDI)*, June 2007.

[14] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults - online proof appendix. Web site: `http://www.cs.princeton.edu/sip/projects/zap/tal_cf/`, 2007.

[15] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[16] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[18] M. A. Schuette and J. P. Shen. Exploiting instruction-level parallelism for integrated control-flow monitoring. In *IEEE Transactions on Computers*, volume 43, pages 129–133, February 1994.

[19] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[20] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, pages 137–143, July 2003.

[21] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *ACM International Conference on Functional Programming*, Portland, Oregon, Sept. 2006.