

Reasoning about Control Flow in the Presence of Transient Faults ^{*}

Frances Perry and David Walker

Princeton University
{frances, dpw}@cs.princeton.edu

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 [10] and Reis [15]. 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.

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, 17, 9], rendering processors that use them more susceptible to *transient faults*. 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 caused significant failures in server farms at companies including AOL and eBay [4] and in supercomputers such as those at Los Alamos Labs [8]. More importantly, current hardware manufacturing trends suggest the problem of transient faults will grow substantially in the future [6].

In order to counter the threat of transient faults, researchers from industry and academia have been searching for solutions to the reliability problem in both

^{*} This research is funded in part by NSF award CNS-0627650 and a Microsoft graduate fellowship. 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.

hardware and software. Broadly speaking, with 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. In terms of cost-effectiveness, recent studies have shown that software techniques for fault tolerance often add approximately 35% overhead [15] 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 [18], 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 [11], 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 hardware state to detect those faults.

Surprisingly, however, researchers [10, 15, 5] have developed techniques for detecting certain classes of control-flow errors entirely in software. These techniques do not catch all control-flow errors, but empirical evidence suggests they can improve system reliability substantially. However, 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 classic while programs into well-typed programs in the language. Due to space limitations, this extended abstract only describes selected elements of our assembly language and its type system. Further details may be found in an accompanying technical report [12] and in our online proofs [13].

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 when a fault effects either (1) the target address of a jump instruction, (2) the target address of a conditional jump instruction, or (3) the boolean condition of a conditional jump instruction. 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 control is either transferred to the beginning of some block or to an illegal instruction, which is immediately detected by the hardware. We currently do not consider the possibility that a fault causes a control flow transfer to a legal instruction in the middle of a block. (For a discussion on alleviating this restriction, see Section 6.) In addition, we adhere to the standard *Single Event Upset* model [14, 16], which states that only one fault may occur during an execution, though faulty values may be copied, propagated and used in any way an ordinary value may be used.

In order to ensure that control flow transfers do not go wrong, compiled code computes a replica of the intended control-flow destination prior to the control-flow transfer and moves 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; ...; brnz r2, Lrecover; ...
```

Here, the current block address `Lk` is loaded into register `r2` and 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`. If the corrupted value is not a valid code address, then a hardware fault will be triggered. Otherwise, upon arrival at some erroneous control flow

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

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. If `ri` is corrupted, 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. 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. 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 example, 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 the `recovernz` instruction will not help detect the fault.

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 the jump target is held in registers `r3` and `r3'` and the condition for the jump is held in registers `r4` and `r4'`. These register pairs must be *independent replicas* of one another. In other words, in the absence of faults, they should contain the same 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 the target in `r3`. It also uses a conditional move `cmovz r4', ri, r3'`, `ri, r3'`, which moves the contents of `r3'` into `ri` if `r4'` is zero, and otherwise does nothing.²

```
L1: ...; intend L2; cmovz r4', ri, r3'; brz r4, r3
```

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.

² Many architectures including the IA-32 following the Pentium Pro, the Sparc V-9 and the IA-64 have conditional moves. Other architectures can use a conditional branch and a move instruction instead, but this branch will not be protected.

<i>colors</i>	$c ::= G \mid B \mid O$	<i>instructions</i> i	$::= \text{movi } r_d \ v$
<i>colored values</i> v	$::= c \ n$		$\mid \text{sub } r_d \ r_s \ r_s$
			$\mid \text{intend } r_t$
<i>code memory</i> C	$::= \cdot \mid C, \ell \rightarrow b$		$\mid \text{intendz } r_z \ r_t$
			$\mid \text{recovernz } r_z$
<i>registers</i>	$r ::= r_i \mid r_1 \mid \dots \mid r_n$	<i>blocks</i>	$b ::= i; b \mid \text{jmp } r_t \mid \text{brz } r_z \ r_t$
<i>register file</i> R	$::= \cdot \mid R, r \rightarrow v$	<i>states</i>	$\Sigma ::= (C, h, R, b)$
<i>history</i>	$h ::= \ell_1, \dots, \ell_n$	<i>final states</i> \mathcal{F}	$::= \Sigma \mid \text{recover}(h) \mid \text{herror}(h)$

Fig. 1. Machine State Syntax.

Summary. By creating duplicate copies of intended control flow targets, it is possible to check that control has arrived at the proper location. 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**. Instruction operands include constant values v and registers r . In the previous section, values were unannotated, but from this point forward we annotate every value with a *color* c where c is either G (green), B (blue) or O (orange). These colors have no operational significance, but they play 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 we use meta-variable ℓ to emphasize that an integer will be used as an address.

Instructions are grouped together in code blocks b that 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 the notation $\ell + 1$ refers to the address of the block following the block at ℓ . If a block at ℓ ends with a conditional branch, $\ell + 1$ must inhabit the domain of C .

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

Static Expressions		Types	
<i>exp kinds</i>	$\kappa ::= \kappa_{int} \mid \kappa_{hist}$	<i>stage description</i>	$\rho ::= check \mid ok$ $\mid go \mid goz$
<i>exp contexts</i>	$\Delta ::= \cdot \mid \Delta, x : \kappa$	<i>basic types</i>	$\tau ::= int \mid \rho \mid \forall[\Delta](\Gamma, \sigma)$
<i>exps</i>	$e ::= x \mid n \mid e - e$ $\mid e?e : e$	<i>value types</i>	$t ::= \langle c, \tau, e \rangle$
<i>substitutions</i>	$S ::= \cdot \mid S, e/x$	<i>type option</i>	$\tau_{opt} ::= \tau \mid undef$
Context Typing		ZapTags	
<i>heap typing</i>	$\Psi ::= \cdot \mid \Psi, \ell \rightarrow \tau$	<i>zap tag</i>	$Z ::= \cdot \mid c \mid CF$
<i>reg file types</i>	$\Gamma ::= \cdot \mid \Gamma, r \rightarrow t$		
<i>history typing</i>	$\sigma ::= \epsilon \mid x \mid \sigma \circ e$		

Fig. 2. Typing Syntax.

of labels. It records the code blocks visited during the current execution. In addition to ordinary abstract machine states, “final states” \mathcal{F} include two special states. The state **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 **herror**(h) represents a state in which a transient fault causes transition to an invalid address. Figure 1 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 $\Sigma \rightarrow_k \mathcal{F}$ where k , which is either zero or one, records the number of faults that occur during the step.

The most interesting rules in the system are the rules modeling faults. The primary rule (*zap-reg*) arbitrarily corrupts the value in a single register, though the color tag (which has no operational significance) remains unchanged.

$$\frac{R(r) = c \ n}{(C, h, R, b) \rightarrow_1 (C, h, R[r \mapsto c \ n'], b)} \text{ (zap-reg)}$$

The rule above may fire at any time, just as a transient fault may occur at any point. In particular, it may fire just prior to execution of a jump (**jmp** r_t) or a branch (**brz** r_z r_t), corrupting the jump target in register r_t .

For uniformity in our fault model, we also consider errors in execution of the **recovernz** r_z instruction. These rules, as well as the rules for normal instruction execution, are provided in the technical report [12].

4 Typing

The overall design of the type system is based on two nonstandard concepts: (1) Classifying the reliability properties of values, and (2) Using abstract types to

make sure that the fault tolerance protocol proceeds in the correct order, with no steps omitted or inappropriate steps inserted. The following paragraphs explain the main intuitions behind each concept.

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. Consequently, as is usual, the type system will have to approximate these properties somehow. It does so by assigning each value a color and ensuring that values with the same color have related reliability properties.

Most values either belong to the green group or to the blue group. These two groups have the property that they are *independent* and *redundant*. 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. However, *orange* values are manipulated in such a way as to preserve 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. 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 “.”), 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 values with color c may have been corrupted, but other values will be correct. The final zap tag CF classifies machine states after a control-flow error has occurred. In this case, we know nothing about green or blue values, but the properties of orange values remain valid. The table below summarizes the properties that hold under each zap tag while in block ℓ . 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). The

table says a value is *untrusted* when the standard canonical forms properties do not necessarily hold.

Zap Tag	<i>G</i> values	<i>B</i> values	<i>O</i> values	ℓ is the intended destination
.	trusted	trusted	trusted	yes
<i>G</i>	<i>untrusted</i>	trusted	trusted	yes
<i>B</i>	trusted	<i>untrusted</i>	trusted	yes
<i>O</i>	trusted	trusted	<i>untrusted</i>	yes
<i>CF</i>	<i>untrusted</i>	<i>untrusted</i>	trusted	<i>no</i>

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.
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.
3. The exit code sequence sets the intended target and transfers control to the new block.

The following subsections elucidate some of the technical ideas behind these intuitions. The complete definitions are described more thoroughly in our technical report [12].

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. The *basic type* τ is either an integer type (*int*), a code type ($\forall[\Delta](\Gamma, \sigma)$), or a special type ρ that indicates the state of the fault tolerance protocol. The *static expression* e describes the value in more detail. These static expressions are used by the expression typing rules to require that blue and green computations compute identical results in the absence of faults. The 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. The judgment $\Delta \vdash e : \kappa$ holds when all free variables in e are contained in the context Δ . 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 . 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 value typing judgment has the form $\Delta; \Psi \vdash^Z v : t$. Here, Δ contains expression variables free in t 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 central rule expresses the fact that if a value n has basic type τ , is equal to e and annotated with color c then it 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. Another key rule expresses the fact that when the zap tag is CF , green and blue values can be given *any* type. In particular green values may be given blue types and vice versa.

The type system also uses a subtyping judgment with the form $\Delta \vdash t \leq t'$. As an example, this judgment allows type $\langle c, \tau, e \rangle$ to be a subtype of $\langle c, int, e' \rangle$ whenever $\Delta \vdash e = e'$.

4.2 Instruction and Block Typing

Figure 3 presents several rules from the key judgments for checking program code. The first judgment 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 types required prior to execution of the instruction. Γ' acts as the post condition 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 updates the type of the destination register 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. Neither rule places any restrictions on the type of r_i , so they can occur during any stage of a block.

Though `intend r_t` is operationally the same as `movi r_i r_t` , its typing rule requires that the intentions register r_i has basic type *ok*. This restriction guarantees any new intend will occur after the checking code has been completed. Notice also that the intention register is marked blue – in contrast, the address used as the real jump target will be marked green. Finally, the type of r_i is updated to reflect the new static expression and the new stage *go*.

A sequence of instructions is typed using the block typing judgment, which has the form $\Delta; \Psi; \Gamma; \sigma; e_i; \tau \text{ opt} \vdash b$. In addition to Δ , Ψ , and Γ , the block typing judgment is parametrized by a sequence σ , an expression e_i , and a type option $\tau \text{ opt}$. The sequence σ contains a list of expressions that describe the locations

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

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

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

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

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

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

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

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

Fig. 3. Selected Instruction Typing Rules and Block Typing Rules.

in the current history h . The expression e_i describes the intended target when the transfer occurred to the current label ℓ . If control flow correctly transferred to ℓ , then $\Delta \vdash e_i = \ell$. The option type $\tau \text{ opt}$ 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. Three example rules are shown in Figure 3.

The first rule, *sequence-t*, is used when the first instruction in a block is one of the basic instructions described previously. The second rule for checking blocks illustrates how to check the **recovernz** instruction. At run time, control only proceeds past this point in the block if x_i (describing r_i) is equal to the expression e_ℓ (describing 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 $\Delta \vdash \Gamma/r_i/r_z \text{ wf}$ and $\Delta \vdash \sigma \text{ wf}$ hold when all expression variables used in the types of registers other than r_i and r_z , as well as in 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.

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 actual jump target in r_t 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 `jmp r_t` instruction recolors the blue intention register to be orange when control is transferred 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.

5 Formal Properties

We have proven a number of properties of our type system including variants of the standard Progress, Preservation and Type Safety theorems. Our most important result is a Fault Tolerance theorem, which we sketch briefly below. The full proofs appear in the online appendix [13].

In order to explain the theorem, we require a couple of additional concepts. First, we say a machine state Σ is well-formed (written $\vdash^Z \Sigma$) when all code and state are well-typed relative to the zap tag Z . Second, we say a faulty machine state Σ_f simulates a fault-free state Σ under color c (written $\Sigma_f \sim^c \Sigma$) whenever the two states are identical modulo values colored c . In other words, values colored c may be completely different from one another, but otherwise the two states are identical.

The judgment $\Sigma \Longrightarrow_k^h \mathcal{F}$ states that machine state Σ executes through a sequence of blocks h to reach state \mathcal{F} while incurring k faulty transitions. So if $\Sigma = (C, h_1, R, b)$, then \mathcal{F} is either $(C, (h_1, h), R', b')$, `herror`(h_1, h), or `recover`(h_1, h).

We say 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 final faulty state simulates the original result 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 visited the same blocks as the original computation. (3) A fault affecting the intentions register or checking code cause the faulty computation to conservatively detect a fault in software and jump to recovery 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 1 (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 $\text{length}(h_f) \leq \text{length}(h)$ fit one of these cases:*

1. $\Sigma \Longrightarrow_1^h \Sigma'_f$ and $\exists c . \Sigma'_f \stackrel{c}{\sim} \Sigma'$
2. $\Sigma \Longrightarrow_1^{h_f} \text{herror}(h', h_f)$ and h_f is a prefix of h
3. $\Sigma \Longrightarrow_1^{h_f} \text{recover}(h', h_f)$ and h_f is a prefix of h
4. $\Sigma \Longrightarrow_1^{h_f} \text{recover}(h', h_f)$ and $h_f = (h_1, l')$ and $h = (h_1, l, h_2)$

6 Related Work, Future Work, and Conclusions

Related Work. As mentioned in the introduction, this research follows previous work on λ_{zap} [18] and TAL_{FT} [11]. However, neither λ_{zap} nor TAL_{FT} provided software mechanisms for guaranteeing control-flow integrity. Recently, Elsmann [7] 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. CFI attackers can modify arbitrary amounts of machine state in arbitrary ways, but cannot touch three reserved registers during the execution of certain code sequences.

Our work builds upon many past research efforts in fault tolerance, particularly those that deal with control-flow checking. For example, Oh *et al.* [10] developed a pure software control-flow checking scheme (CFCSS) wherein each control transfer generates a run-time signature that is validated by error checking code generated by the compiler for every block. The SWIFT system [15], another software-only fault tolerance system, also uses signature checking very much like that in the current paper. 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.

Future Work. We acknowledge that the fault model used in this paper is simplistic. By assuming hardware support to catch control transfers into the middle of blocks, we avoid dealing with many interesting and likely situations. This

assumption is required because stating intentions involves resetting r_i , so an incorrect transfer into a block before the `intend` r_t instruction may not be caught.

A sequence of existing work on software-only solutions [10, 15, 5] handles increasing classes of erroneous transfers. By ensuring that the intentions register is a function of the entire control-flow path, not just the current block, they can detect most jumps into the middle of blocks. For example, SWIFT [15] keeps an "approximate program counter" which contains the current block. Before each control-flow transfer, the current block and the intended target block are xored together and put in a designated "transition register". At the beginning of each block, the transition register is xored with the approximate program counter to give the new approximation. (A correct transfer from block A to block B will result in a new approximate program counter of $A \otimes (A \otimes B)$, which is equal to B .) Though these solutions are an improvement, there are still situations (such as jumping back two instructions within a block) that they cannot handle.

The classification scheme of values and reliability properties from this paper does not transfer directly to these more complex solutions, but we believe we can develop a similar classification to capture the necessary invariants. In doing so, the Fault Tolerance Theorem becomes more difficult to state and prove due to the increase of possible scenarios a single fault may cause. (For example, a fault may cause control to transfer from the middle of one block to the middle of a second block. This second block may transfer control to a third block before the error is finally detected.) In essence, the current work and proof strategy are an important building block for reasoning about more complex solutions.

Conclusions. Future processors will become more susceptible to transient faults, and 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. We have rigorously proven strong reliability properties for our type system and believe this is the first successful attempt at reasoning rigorously about software mechanisms for controlling control flow faults.

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. E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.
6. 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.
7. M. Elsmann. 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.
8. 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 Laboratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.
9. 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.
10. 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.
11. 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.
12. F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. Technical Report TR-799-07, Princeton University, 2007.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *ACM International Conference on Functional Programming*, Portland, Oregon, Sept. 2006.