# Faulty Logic: Reasoning about Fault Tolerant Programs

Matthew L. Meola and David Walker

Princeton University, Computer Science Department,
35 Olden Drive, 08540-5233 Princeton, New Jersey
{mmeola,dpw}@princeton.edu

**Abstract.** Transient faults are single-shot hardware errors caused by high energy particles from space, manufacturing defects, overheating, and other sources. Such faults can be devastating for security- and safety-critical systems. In order to mitigate these problems, software developers can add redundancy in various ways to their software systems. However, such redundancy is hard to reason about and corner cases are easy to miss, leaving these systems vulnerable. To solve this problem, we have developed a logic, based on Separation Logic, for reasoning about faults as resources. We show how to use this logic as a language of assertions and incorporate it into a Hoare Logic for verifying imperative programs. This Hoare Logic is parameterized by a formal fault model and it can be used to prove imperative programs correct with respect to that model. In addition to developing this basic verification platform, we have designed a modal operator that abstracts away the effects of individual faults, enabling modularization of proofs and greatly simplifying the reasoning involved. The logic is proved sound and studied through a number of examples, including a simplified version of the RSA Sign/Verify algorithm.

## 1 Introduction

Programmers almost always implement software under the assumption that the underlying hardware is completely reliable. This is the right choice – implementing software correctly is hard enough without worrying about hardware reliability. Nevertheless, there are a number of important situations in which a software engineer must face the fact that hardware faults can and do occur.

One such domain involves the implementation of cryptographic algorithms. For years, software engineers assumed that, while faults in these algorithms might occur, they would not reveal anything important about the embedded cryptographic secrets. However, in 1997, Boneh, DeMillo and Lipton [1] showed how a single fault in common implementations of RSA could be exploited to discover the underlying secret key. Moreover, since that time, other researchers have uncovered problems in DES, RC5 and AES. In related work, Govindavajhala and Appel showed how to exploit faults to break into a commercial Java virtual machine running completely type safe code [2]. There is currently a rich community dedicated to researching these threats and developing solutions. Bar-El's survey paper [3], provides an excellent overview of the area.

In addition to worrying about faults in security-sensitive contexts, engineers must also consider their ramifications when fully optimizing systems for power and performance. For example, by decreasing hardware voltages one can save power at the expense of occasionally incurring faults, and by overclocking one can speed up performance, again at the expensive of the occasional erroneous result. Hedge and Shanbhag [4] illustrate the advantages of exploiting such tradeoffs in digital signal processing applications. Other contexts in which intermittent hardware faults have a significant overall impact may include safety-critical applications, avionics, satellites, supercomputers, and long-running simulations or experiments.

In situations such as these, conventional techniques for reasoning about programs are no longer sound. Consequently, we have begun to develop a new framework that will allow programmers to prove strong properties about their programs despite the presence of faults. Our framework involves a relatively simple and self-contained extension to a standard Hoare Logic for while programs. This extension allows programmers to reason about the faults that may or may not have happened to their programs in typical Hoare style. Transient faults appear explicitly as objects in the logic, and operators inspired by Separation Logic are used to count, limit, and contain the faults.

In summary, the main contributions of the paper are: the development of a logic for proving programs to be fault tolerant, the proof of soundness for this logic, parameterization of the logic by one of multiple fault models, illustration of logic's use through examples in multiple application areas, the proof that the logic supports the frame rule, the development of a modality that supports concise proofs, and a weakest precondition Hoare rule for the extension of Hoare Logic.

The rest of the paper is organized as follows. Section 3 discusses the programming language, including a new instruction, `fault`, which introduces the possibility of a fault at a specific program point. Section 4 extends standard Hoare Logic with the rule for `fault`. Section 5 demonstrates the complexity of dealing with fault functions explicitly in proofs and introduces a modality that abstracts away the explicit fault functions. Section 6 illustrates the application of the logic in security protocols, through a specification for a fault tolerant implementation of the RSA Sign/Verify protocol. Section 7 describes a compilation from programs and specifications in standard Hoare Logic into programs in our logic with fault tolerance achieved through triple modular redundancy. Related work is discussed in Section 8, and Section 9 concludes.

## 2 Modeling Faults

Before we can reason about faults, and indeed before programmers or hardware designers can protect against faults, there must be some kind of model for when and where faults can occur. Typical fault models dealt with in the literature are fairly simple, limiting faults to one or a few occurrences per program run. The most common models are the Single Event Upset (SEU) and Single Word

Corruption (SWC) models. The SEU model allows a single bit flip in a single register in one run of the program, as seen in the work of Chang, Reis, and August; Shirvani, Saxena, and McCluskey; Bar-El, et al.; among others [5, 3, 6]. The SWC model allows arbitrary changes to a single register to occur once in the program, as seen in Bar-El, et al. and Shirvani, Saxena, and McCluskey [3, 6]. The motivation behind these fault models is twofold: one, that the incidence of faults is rare enough that programmers may ignore the negligible chance of two occurring; and two, that the fault model defines a class of errors that is possible to protect against without extreme performance degradation. For this reason, we mainly focus on these two fault models. However, our logic supports other fault models, including those allowing up to two faults to occur during a single program run. Such a model is briefly examined in this paper.

## 3 The Programming Language

The programming language that we consider in this paper is the classic imperative language of while programs extended with a single pseudo-instruction that is used to specify where faults may occur within a program. For example, consider a simple loop:

```
x := 0;
while x != 0 do
  skip;
```

Here, the program variable `x` is assigned zero and the program loops endlessly, testing `x` for inequality with zero. To reason about the execution of the program in the presence of faults, the programmer or a static analysis inserts fault statements at appropriate program points. For example:

```
x := 0; fault x;
while x != 0 do
  { skip; fault x; }
```

This allows faults to occur at two points in the program. Intuitively, the statement `fault x` means that a fault *may occur* to program variable `x` at this point in the computation. Hence, by inserting the `fault x` statement between every pair of lines, the programmer considers the possibility that faults may occur at any point in the program. [1] Thus, the programming language and the logic to be introduced later in the paper are agnostic about where faults may occur in the program. This allows the programmer to focus on protecting critical sections of code.

If there are multiple program variables, each program variable must be mentioned separately. For example:

[1] The reader may note that in any fault model where any occurring fault is arbitrary (such as the SWC model, or an n-word corruption model), it suffices to introduce a `fault` statement for a variable `x` immediately before each time the variable's value is read. This is also true for any fault model allowing at most one fault (including both the SWC and SEU models).

```
x := 0; fault x;
y := 0; fault y;
while (x != 0) and (y != 0) do
  { skip; fault x; fault y; }
```

To abbreviate long sequences of fault statements, we normally write `fault` $\texttt{x}_1$`,...,`$\texttt{x}_n$`;` in place of `fault` $\texttt{x}_1$`; ... fault` $\texttt{x}_n$`;`.

The observant reader will also notice that there is no syntax for faults that may occur in the midst of a complex expression in a while loop bound, if statement, or right-hand side of an assignment. To consider such faults, the programmer must decompose the expressions into a series of statements:

```
x := 0; fault x;
y := 0; fault y;
flag1 := x != 0; fault flag1;
flag2 := y != 0; fault flag2;
flag1 := flag1 and flag2; fault flag1, flag2;
while flag1  do
  { skip; fault x, y, flag1, flag2;
     flag1 := x != 0; fault flag1;
     flag2 := y != 0; fault flag2;
     flag1 := flag1 and flag2; fault flag1, flag2;
   }
```

This example makes it clear that as programs get more complex, there is a proliferation of fault instructions. On the one hand, this proliferation reveals the inherent difficulty of reasoning about programs in a context with a rich fault model. On the other hand, it demonstrates that a production verification system should probably manage the insertion of fault instructions itself (e.g., by having the static analysis engine insert them automatically). In this paper, we leave the fault instructions in the syntax of the programming language because doing so makes the formal development particularly clear, modular, and self-contained. In a production environment, this language would correspond to an intermediate language or a language used with a proof assistant.

### 3.1 Syntax

A summary of the syntax of the language we use in the paper is presented in Figure 1. Here and throughout the rest of the paper, we let $x$ range over program variable names, $n$ range over integers and $f$ range over computable functions from integers to integers. The specific set of integer and boolean expressions we choose for the language is unimportant and hence we will freely use other expressions in our examples as they require. Note that function variables do not appear in the source language itself. They are only used in expressions that appear in the program logic, to be described later.

$$
\begin{array}{lll}
\text{integer vars} & x & \qquad\qquad\text{integers} \quad n \\
\text{function vars} & \phi & \qquad\qquad\text{functions} \quad f \\
\text{function exps} & G & ::= \phi \mid f \\
\text{integer exps} & E & ::= x \mid n \mid E_1 + E_2 \mid E_1 \texttt{ mod } E_2 \mid G\ E \mid e \\
\text{boolean exps} & B & ::= E_1 = E_2 \mid \texttt{not } B \mid B_1 \texttt{ and } B_2 \mid E_1 < E_2 \\
\text{statements} & S & ::= \texttt{skip} \mid x := E \mid S_1; S_2 \mid \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \\
 & & \ \ \mid\ \ \texttt{while } B \texttt{ do } S \mid \texttt{fault } x
\end{array}
$$

**Fig. 1.** Syntax of Programs

### 3.2 Representation of Faults and Fault Models

When a `fault x` statement is executed, the value of `x` may change. Such changes can be represented by a function, $f$, on the integers. The function acts on the variable `x`, causing the new value, $f$ `x`, to be stored there. For example, if the third bit of `x` is flipped, the function a bit flip function, written $\lambda y.y \texttt{ xor } 2^2$ as a lambda expression [2], will represent this fault. Similarly, if `x` is unchanged, the identity function will represent this trivial fault.

Over the course of a program run, we record the fault functions that have occurred in the *fault state* but not the variables that they applied to. This is because the effects of a fault spread wider than the initial variable affected and we are not doing any calculations of information flow to track the effects. Formally, fault states ($F$) are multi-sets and we use the notation $F_1 + F_2$ to denote multiset union of fault states. We also write $F_1 \subseteq F_2$ when $F_1$ is a sub-multiset of $F_2$. As an example, the fault state $\{\lambda x.x \texttt{ xor } 2^3\}$ represents a situation in which a single fault has occurred and that fault has toggled the 4th bit of the associated value. Over the course of a run, it is common for many trivial faults to occur and this will lead to an accumulation of identify functions in the fault state. For instance, the fault state $\{\lambda x.x \texttt{ xor } 2^3, \lambda x.x, \lambda x.x, \lambda x.x\}$ represents a situation in which only one true fault has occurred, but three additional trivial faults have been recorded in the fault state. [3]

A judgment $F\ \mathsf{ok}_m$ defines the fault states $F$ that are allowed by the fault model $m$. Most of the rest of our development is independent of the particular choice of fault model except for the restrictions that the empty fault state must be valid and that validity must be preserved by subset ordering.

**Definition 1 (Fault State Validity Criterion).**

- $\{\}\ \mathsf{ok}_m$.
- *If* $F_1\ \mathsf{ok}_m$ *and* $F_2 \subseteq F_1$ *then* $F_2\ \mathsf{ok}_m$.

Using multisets of functions as our fault states is elegant and easy to work with and yet allows us to reason about several different interesting fault models.

[2] Note that mathematical functions, not lambda expressions, are part of our logic, Lambda expressions are just used as a convenient representation.

[3] Allowing the fault state to accumulate many trivial faults helps simplify our operational semantics slightly.

In this paper, we will work with the following three fault models, each of which maybe characterized according to its $F$ $\mathtt{ok}_m$ relation, though the bulk of our work should extend to related models. The models are characterized by their $F$ $\mathtt{ok}_m$ relations, each of which satisfies the Fault State Validity Criterion.

**Definition 2 (SWC Fault Model).** *The SWC fault model demands that* $F$ $\mathtt{ok}_m$ *if and only if at most one function* $f$ *drawn from* $F$ *is not the identity function.*

**Definition 3 (SEU Fault Model).** *The SEU fault model demands that* $F$ $\mathtt{ok}_m$ *if and only if at most one function* $f$ *drawn from* $F$ *is not the identity function and that non-identity function* $f$ *has the form* $\lambda x.x$ $\mathtt{xor}$ $2^k$ *for some* $k$*.*

**Definition 4 (DWC Fault Model).** *The DWC fault model demands that* $F$ $\mathtt{ok}_m$ *if and only if at most two functions* $f$ *and* $g$ *drawn from* $F$ *are not the identity function.*

### 3.3 Operational Semantics

A program state is a triple $(F, V, Z)$ where $F$ is the current fault state, $V$ is the current environment and $Z$ is either a statement $S$ to execute or $-$, indicating execution is complete. We call states with the form $(F, V, -)$ *final states*. An environment is a finite partial map from variable names to integer values. We write $V(x)$ to denote the contents of the map at $x$ and we write $V[x \mapsto n]$ to denote the map created by updating $V$ at $x$ with $n$.

The operational semantics of the language are presented in Figure 2. These rules depend upon a conventional denotational semantics (see, for example, Winskel, Chapter 5 [7]), which, given an environment, maps integer expressions to integers and boolean expressions to 0 (false) or 1 (true). We write the semantic functions $[\![E]\!]_V$ and $[\![B]\!]_V$ respectively.

The rules governing the standard statements (`skip`, assignment, `if`, and `while`) leave the fault state untouched and behave in the usual way. The operational rule for the fault statement non-deterministically chooses a fault function $f$ that satisfies the given fault model, transforms the contents of the given variable, and adds $f$ to the fault state. Note that $f$ may be the identity function, meaning that a fault statement indicates a program point where a fault *may* occur as opposed to where a fault must occur.

## 4 The Program Logic

Having described our programming language, we now present the programmer with the tools to reason about these programs. These tools consist of a basic Hoare Logic with extensions to allow reasoning about faults in program variables.

As a reminder, a Hoare triple is written $\{P\}S\{Q\}$. Following the rules of partial correctness, the Hoare triple means that, if $P$ describes the program state immediately before $S$ is executed and the execution of $S$ terminates, then $Q$ will describe the resulting program state.

$$\text{Eskip}\,\frac{}{(F, V, \texttt{skip}) \longmapsto (F, V, -)}$$

$$\text{Eassign}\,\frac{}{(F, V, x := E) \longmapsto (F, V[x \mapsto [\![E]\!]_V], -)}$$

$$\text{Eseq1}\,\frac{(F, V, S_1) \longmapsto (F', V', S_1')}{(F, V, S_1; S_2) \longmapsto (F', V', S_1'; S_2)}$$

$$\text{Eseq2}\,\frac{(F, V, S_1) \longmapsto (F', V', -)}{(F, V, S_1; S_2) \longmapsto (F', V', S_2)}$$

$$\text{Eif1}\,\frac{[\![B]\!]_V = 1}{(F, V, \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \longmapsto (F, V, S_1)}$$

$$\text{Eif2}\,\frac{[\![B]\!]_V = 0}{(F, V, \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \longmapsto (F, V, S_2)}$$

$$\text{Ewhile1}\,\frac{[\![B]\!]_V = 0}{(F, V, \texttt{while}\ B\ \texttt{do}\ S) \longmapsto (F, V, -)}$$

$$\text{Ewhile2}\,\frac{[\![B]\!]_V = 1}{(F, V, \texttt{while}\ B\ \texttt{do}\ S) \longmapsto (F, V, S; \texttt{while}\ B\ \texttt{do}\ S)}$$

$$\text{Efault}\,\frac{F + \{f\}\ \mathsf{ok}_m}{(F, V, \texttt{fault}\ x) \longmapsto (F + \{f\}, V[x \mapsto f\ (V(x))], -)}$$

**Fig. 2.** Operational Semantics of Programs

Figure 3 contains inference rules and assertion language for a basic Hoare Logic, with a subscript $m$ added for use in our logic. The subscript refers to the fault model considered in the Hoare triples. Note that the assignment rule works backwards. If some assertion $P$ describes the program state after the assignment of $E$ to $x$, then the same assertion with all occurrences of $x$ replaced with $E$ describes the state before the assignment.

### 4.1 A Straw Man Logic

Before describing our actual Hoare Logic, it is instructive to consider why a naive extension of our basic Hoare Logic does not work. Taking a cue from the assignment rule, we could generate a precondition from a postcondition by replacing the affected variable with the value it is assigned by the statement.

$$\text{Hfault} - \text{try1}\,\frac{}{\{P[f\ x/x]\}\texttt{fault x}\{P\}_m}$$

seems to be a plausible start, as the operational semantics say that the value of $x$ changes to $f\ x$ for some function $f$. In order to consider all possible faults, we quantify over all possible functions on the integers:

$$\text{Hfault} - \text{try2}\,\frac{}{\{\forall\phi.\ P[\phi\ x/x]\}\texttt{fault x}\{P\}_m}$$

$$\text{Hskip}\;\frac{}{\{P\}\texttt{skip}\{P\}_m}$$

$$\text{Hassign}\;\frac{}{\{P[E/x]\}\texttt{x := E}\{P\}_m}$$

$$\text{Hwhile}\;\frac{\{B \mathbin{\&} P\}S\{P\}_m}{\{P\}\texttt{while } B \texttt{ do } S\{\neg B \mathbin{\&} P\}_m}$$

$$\text{Hif}\;\frac{\{B \mathbin{\&} P\}S_t\{Q\}_m \quad \{\neg B \mathbin{\&} P\}S_e\{Q\}_m}{\{P\}\texttt{if } B \texttt{ then } S_t \texttt{ else } S_e\{Q\}_m}$$

$$\text{Hcons}\;\frac{P' \vDash_m P \quad \{P\}S\{Q\}_m \quad Q \vDash_m Q'}{\{P'\}S\{Q'\}_m}$$

$$\text{Hseq}\;\frac{\{P\}S_1\{Q\}_m \quad \{Q\}S_2\{R\}_m}{\{P\}S_1\,;S_2\{R\}_m}$$

$$P ::= \mathsf{true} \mid \mathsf{false} \mid \neg P \mid E = E \mid \forall x.P \mid \exists x.P \mid P \vee P \mid P \mathbin{\&} P$$

**Fig. 3.** Inference Rules and Assertion Language for a basic Hoare Logic

Unfortunately, this rule does not integrate any properties of the fault model. This makes the rule quite useless, as the following example [4] using the SWC fault model, *m*, demonstrates:

*Example 1.*

$$\begin{array}{ll}
 & \{\mathsf{false}\} \\
 & \{\forall \phi_1, \phi_2.\ \phi_1\ 3 = 3 \vee \phi_2\ 3 = 3\}_m \quad \text{(equivalent)} \\
\texttt{x = 3;} & \{\forall \phi_1, \phi_2.\ \phi_1\ x = 3 \vee \phi_2\ 3 = 3\}_m \\
\texttt{y = 3;} & \{\forall \phi_1, \phi_2.\ \phi_1\ x = 3 \vee \phi_2\ y = 3\}_m \\
\texttt{fault x,y;} & \{x = 3 \vee y = 3\}_m
\end{array}$$

Under the SWC fault model, at least one of the variables should equal 3 at the end, no matter what state the program begins in. However, the precondition we derive is equivalent to false and thus not true in any state. The problem is that our candidate Hoare rule does not allow us to apply any information about the fault model to the assertions. We need a way to describe the fault functions that can actually occur in the fault state.

### 4.2 A Useful Logic

The key insight is that we need a predicate $\mathsf{hap}\, f$ ("$f$ happened") that says that a fault function is in the current fault state. $\mathsf{hap}\, f$ is true whenever the fault

[4] In our examples, the left column contains code and the right column contains the corresponding assertions. A line of code, the precondition above and to the right, and the postcondition to the right together form a valid Hoare triple. Assertions one on top of the other with no code to the left indicate entailment. Using the sequence and consequence rules, a sequence of such entailments and Hoare triples results in a valid Hoare triple for the entire example.

function $f$ is the identity or is in the fault state. For example, $\mathsf{hap}\,\lambda x.x$ describes any program state and $\mathsf{hap}\,f$ describes any state where $f$ is in the fault state. This will allow us to reason about fault functions that are allowed in the current fault state.

In order to refer to the addition of fault functions to the state, rather than just their presence, we borrow $-\!\!*$ from Separation Logic [8, 9]. $P -\!\!* Q$ means that, in any state under which $P$ holds, adding that state to the current state makes $Q$ true. For example, $\mathsf{hap}\,f -\!\!* Q$ implies that adding $f$ to the current state makes $Q$ true.

Using both $-\!\!*$ and $\mathsf{hap}$ , we can limit the range of fault functions to those that are allowed in the current fault state.

$$\text{Hfault}\frac{}{\{\forall\phi.\ \mathsf{hap}\,\phi -\!\!* P[\phi\ x/x]\}\mathtt{fault\ x}\{P\}_m}$$

This is the correct Hoare rule for `fault x`. Intuitively, it means that we know $P$ after a fault statement if $P[f\ x/x]$ was true for any allowable fault function $f$ beforehand.

Before we can use the fault rule to reason about the example from the previous section, we need a way to describe the values of fault functions. A simple approach suffices: we introduce predicates to say whether a function $f$ is the identity ($\mathsf{id}\,f$) or not ($\mathsf{faulty}\,f$). For example, $\mathsf{id}\,\lambda x.x$ & $\mathsf{faulty}\,(\lambda x.x\ \mathtt{xor}\ 2^4)$ is always true.

Using the predicates $\mathsf{id}\,f$, $\mathsf{faulty}\,f$, and $\mathsf{hap}\,f$, we can write down simple axioms that characterize our fault models. For instance, we can characterize the SWC fault model through the following axiom. This axiom uses Separation Logic's separating conjunction $P * Q$ to express the fact that both $P$ and $Q$ are true and that they describe disjoint subsets of the fault state.

$$\forall\phi_1,\phi_2.\ \mathsf{hap}\,\phi_1 * \mathsf{hap}\,\phi_2 -\!\!* (\mathsf{id}\,\phi_1 \vee \mathsf{id}\,\phi_2)$$

This axiom says that, of any two fault functions in the fault state, at least one is the identity [5]. The separating conjunction in $\mathsf{hap}\,\phi_1 * \mathsf{hap}\,\phi_2$ guarantees that $\phi_1$ and $\phi_2$ do not refer to the same fault function instance in the fault state.

Using the proper Hoare rule for fault and this axiom about the SWC fault model, the example from the previous section works perfectly.

*Example 2.*

$\{\mathsf{true}\}_m$

$\{\forall\phi_2,\phi_1.\ \mathsf{hap}\,\phi_2 * \mathsf{hap}\,\phi_1 -\!\!* \mathsf{id}\,\phi_1 \vee \mathsf{id}\,\phi_2\}_m$ (by above property)

$\{\forall\phi_2,\phi_1.\ \mathsf{hap}\,\phi_2 * \mathsf{hap}\,\phi_1 -\!\!* \phi_1\ 3 = 3 \vee \phi_2\ 3 = 3\}_m$

$\{\forall\phi_2.\ \mathsf{hap}\,\phi_2 -\!\!* \forall\phi_1.\ \mathsf{hap}\,\phi_1 -\!\!* \phi_1\ 3 = 3 \vee \phi_2\ 3 = 3\}_m$

`x = 3; y = 3;` $\{\forall\phi_2.\ \mathsf{hap}\,\phi_2 -\!\!* \forall\phi_1.\ \mathsf{hap}\,\phi_1 -\!\!* \phi_1\ x = 3 \vee \phi_2\ y = 3\}_m$

`fault x,y;` $\{x = 3 \vee y = 3\}_m$

[5] The reader may note that the two fault functions added to the fault state in the antecedent of this axiom are not "used" in the consequent. This is allowed, since, as can be seen in Section 4.3, our logic is an affine logic rather than a linear logic such as Separation Logic.

The SEU fault model allows for even more powerful properties, such as:

$$\forall f, x. f\ x \neq x (\text{mod } 3) \text{ iff } \mathsf{faulty}\ f$$

which says that if there is a single bit flip in a variable (the only fault allowed in the SEU model), then difference between the changed variable and its original value is not divisible by 3, as it is a power of 2.

We use this property to prove that a simple example using an AN code is fault tolerant [5]. An AN code is a fault tolerant encoding of integers. To encode an integer encoded in base two, it is multiplied by a number that is relatively prime to two (in this case three). This way, any legal code word is a multiple of three. Any bit single flip will result in a number that is not a multiple of three and thus can be detected. What makes this code so useful is that it commutes with addition:

$$3 \cdot (a + b) = 3a + 3b.$$

This way, additions can be done efficiently on encoded numbers with regular hardware and the results can be checked for errors.

In Figure 4, we show that when using an AN code, only two independent copies of a computation are required to recover from a single bit flip fault, assuming no faults during the recovery code. The example code simply sets the variable `y` to be three times its initial value (while `x` remains at the same initial value). It then loops, waiting for a fault. The code checks whether the fault occurred in `x` or `y` and sets the faulty variable from the unaffected one.

Note that this example uses the standard Separation Logic frame rule

$$\text{Hfaultframe}\ \frac{\{P\}\texttt{fault x}\{Q\}_m \quad \texttt{x} \notin \text{fv}(R)}{\{P * R\}\texttt{fault x}\{Q * R\}_m}$$

which we will prove later. The frame rule allows modular reasoning—if an unrelated assertion is separated from the one currently being considered, then it is unaffected. This is very useful in proofs of many fault tolerance properties including those involving independent redundant computations.

Our logic can also be used with a fault model allowing two arbitrary faults in a single program run. This results in an axiom very similar to that we had for the SWC model. The axiom appears below.

$$\forall \phi_1, \phi_2, \phi_3.\ \mathsf{hap}\ \phi_1 * \mathsf{hap}\ \phi_2 * \mathsf{hap}\ \phi_3 \mathrel{-\!\!*} (\mathsf{id}\ \phi_1 \vee \mathsf{id}\ \phi_2 \vee \mathsf{id}\ \phi_3)$$

Except for the addition of a third assignment and thus a third fault function, the example proceeds exactly like Example 1.

*Example 3.*

$$\begin{array}{ll}
 & \{\mathsf{true}\}_m \\
 & \{\forall \phi_3, \phi_2, \phi_1.\ \mathsf{hap}\ \phi_3 * \mathsf{hap}\ \phi_2 * \mathsf{hap}\ \phi_1 \mathrel{-\!\!*} \mathsf{id}\ \phi_1 \vee \mathsf{id}\ \phi_2 \vee \mathsf{id}\ \phi_3\}_m \\
 & \quad \text{(by the above axiom)} \\
 & \{\forall \phi_3, \phi_2, \phi_1.\ \mathsf{hap}\ \phi_3 * \mathsf{hap}\ \phi_2 * \mathsf{hap}\ \phi_1 \mathrel{-\!\!*} \phi_1 1 = 1 \vee \phi_2 1 = 1 \vee \phi_3 1 = 1\}_m \\
 & \{\forall \phi_3.\ \mathsf{hap}\ \phi_3 \mathrel{-\!\!*} \forall \phi_2.\ \mathsf{hap}\ \phi_2 \mathrel{-\!\!*} \forall \phi_1.\ \mathsf{hap}\ \phi_1 \mathrel{-\!\!*} \phi_1 1 = 1 \vee \phi_2 1 = 1 \vee \phi_3 1 = 1\}_m \\
\texttt{x=3;y=3;z=3} & \{\forall \phi_3.\ \mathsf{hap}\ \phi_3 \mathrel{-\!\!*} \forall \phi_2.\ \mathsf{hap}\ \phi_2 \mathrel{-\!\!*} \forall \phi_1.\ \mathsf{hap}\ \phi_1 \mathrel{-\!\!*} \phi_1 x = 1 \vee \phi_2 y = 1 \vee \phi_3 z = 1\}_m \\
\texttt{fault x,y,z} & \{x = 1 \vee y = 1 \vee z = 1\}_m
\end{array}$$

```
                 $\{x = n * y = n\}_m$
y = 3*y;         $\{x = n * y = 3n\}_m$
                 $\{\exists g_1, g_2.\mathsf{hap}\, g_1 * \mathsf{hap}\, g_2 * y = g_2(3n) * x = g_1 n\})$
while (y=3x)       $\{y = 3x \,\&\, (\exists g_1, g_2.\mathsf{hap}\, g_1 * \mathsf{hap}\, g_2 * y = g_2(3n) * x = g_1 n)\}_m$
  do               $\{(y = 3x \,\&\, y = 3n) \vee (y = 3x \,\&\, x = n)\}_m$
                   $\{y = 3n \,\&\, x = n\}$
                   $\{\forall g_3, g_4.\mathsf{hap}\, g_3 * \mathsf{hap}\, g_4 \mathrel{-\!\!*} \mathsf{hap}\, g_3 * \mathsf{hap}\, g_4 * g_3 y = g_3(3n) * g_4 x = g_4 n\}_m$
                   $\{\forall g_3, g_4.\mathsf{hap}\, g_3 * \mathsf{hap}\, g_4 \mathrel{-\!\!*} \exists g_1, g_2.\mathsf{hap}\, g_1 * \mathsf{hap}\, g_2 *$
                     $g_3 y = g_2(3n) * g_4 x = g_1 n\}_m$
  fault x,y;       $\{\exists g_1, g_2.\mathsf{hap}\, g_1 * \mathsf{hap}\, g_2 * y = g_2(3n) * x = g_1 n)\}_m$
                 $\{y \neq 3x \,\&\, (\exists g_1, g_2.\mathsf{hap}\, g_1 * \mathsf{hap}\, g_2 * y = g_2(3n) * x = g_1 n)\}_m$
                 $\{(y \bmod 3 = 0 \,\&\, y = 3n) \vee (y \bmod 3 \neq 0 \,\&\, x = n)\}_m$
if (y mod 3=0)     $\{y \bmod 3 = 0 \,\&\, ((y \bmod 3 = 0 \,\&\, y = 3n) \vee (y \bmod 3 \neq 0 \,\&\, x = n))\}_m$
then               $\{y/3 = n\}_m$
  y = y/3;         $\{y = n\}_m \vDash_m \{y = n * y = n\}_m$
  x = y;           $\{x = n * y = n\}_m$
else               $\{y \bmod 3 \neq 0 \,\&\, ((y \bmod 3 = 0 \,\&\, y = 3n) \vee (y \bmod 3 \neq 0 \,\&\, x = n))\}_m$
                   $\{x = n\}_m$
                   $\{x = n * x = n\}_m$
  y = x;           $\{x = n * y = n\}_m$
                 $\{x = n * y = n\}_m$
```

**Fig. 4.** Proving a use of AN codes to be fault tolerant under the SEU fault model, $m$

### 4.3 Formal Assertion Semantics

The assertions of our Hoare Logic are based on those of the Separation Logic of Ishtiaq, O'Hearn, and Reynolds [8, 9] with the current fault state taking on the role that the heap has in Separation Logic.

Assertion semantics are defined according to a judgment $F; V \vDash_m P$ between a fault model $m$, well-formed fault state, an environment, and an assertion. This judgment is defined in Figure 5. Note that these semantics depend on the definition of the well-formedness judgment $F\ \mathsf{ok}_m$, which varies according to the fault model being considered. The novelty of these assertions lies in the interaction of the atomic assertions with the Separation Logic connectives $*$ and $\mathrel{-\!\!*}$.

The fault state directly affects only the atomic assertion $\mathsf{hap}\, f$, as the assertions $\mathsf{faulty}\, f$ and $\mathsf{id}\, f$ depend only on the function $f$, and the equality assertion between expressions depends on the environment but not the fault state. Furthermore, the logic is affine: the $\mathsf{hap}\, f$ assertion uses up an occurrence of the function $f$ in the fault state, but the function's appearance in the fault state does not *require* that it is used by a $\mathsf{hap}\, f$. Thus the predicates describe a subset of all elements of the fault state (and possibly additional identity functions).

The purpose of the separating implications is to reason about adding fault functions to states. The separating conjunctions allow reasoning about fault functions that are distinct elements of the fault state. With $\mathrel{-\!\!*}$ we can capture the notion of adding a fault function to the fault state. For example, $F; V \models_m \mathsf{hap}\, f \mathrel{-\!\!*} P$ says that $P$ holds if $f$ is added to the fault state (more precisely, in any fault state containing $F$ plus a copy of $f$). Similarly, $*$ allows us to reason about multiple separate fault functions. The statement $F; V \models_m \mathsf{hap}\, f * \mathsf{hap}\, g \mathrel{-\!\!*} \mathsf{id}\, f \vee \mathsf{id}\, g$ says that if two fault functions are added to the fault state, then at least one of them is the identity. This statement holds under the SWC fault model.

Unlike the heap contents in Separation Logic, fault functions do not refer to one another and there is no way to modify fault functions in our logic. As such, the complex descriptions of heap structure in Separation Logic have no analogue here. This is a good thing, as the large number of fault functions corresponding to possible faults are complex enough.

| $F; V \models_m P$ | |
|---|---|
| $F; V \models_m \forall x.\ P$ | iff $F$ $\mathsf{ok}_m$ and for all $n$, $F; V \models_m P[n/x]$ |
| $F; V \models_m \exists x.\ P$ | iff $F$ $\mathsf{ok}_m$ and there exists $n$ such that $F; V \models_m P[n/x]$ |
| $F; V \models_m \forall \phi.\ P$ | iff $F$ $\mathsf{ok}_m$ and for all $f$, $F; V \models_m P[f/\phi]$ |
| $F; V \models_m \exists \phi.\ P$ | iff $F$ $\mathsf{ok}_m$ and there exists $f$ such that $F; V \models_m P[f/\phi]$ |
| $F; V \models_m \mathsf{hap}\, f$ | iff $F$ $\mathsf{ok}_m$ and $f \in F$ or $f = \lambda x.x$ |
| $F; V \models_m \mathsf{id}\, f$ | iff $F$ $\mathsf{ok}_m$ and $f = \lambda x.x$ |
| $F; V \models_m \mathsf{faulty}\, f$ | iff $F$ $\mathsf{ok}_m$ and $f \neq \lambda x.x$ |
| $F; V \models_m P_1 * P_2$ | iff $F$ $\mathsf{ok}_m$ and there exist $F_1$ and $F_2$ such that $F = F_1 + F_2$, $F_1; V \models_m P_1$, and $F_2; V \models_m P_2$ |
| $F; V \models_m P_1 \mathrel{-\!\!*} P_2$ | iff $F$ $\mathsf{ok}_m$ and for all $F'$, if $F + F'$ $\mathsf{ok}_m$ and $F'; V \models_m P_1$, then $F + F'; V \models_m P_2$ |
| $F; V \models_m E_1 = E_2$ | iff $F$ $\mathsf{ok}_m$ and $[\![E_1]\!]_V = [\![E_2]\!]_V$ |
| $F; V \models_m P_1 \vee P_2$ | iff $F$ $\mathsf{ok}_m$ and $F; V \models_m P_1$ or $F; V \models_m P_2$ |
| $F; V \models_m P_1 \,\&\, P_2$ | iff $F$ $\mathsf{ok}_m$ and $F; V \models_m P_1$ and $F; V \models_m P_2$ |
| $F; V \models_m \neg P$ | iff $F$ $\mathsf{ok}_m$ and $F; V \not\models_m P$ |
| $F; V \models_m \mathsf{true}$ | iff $F$ $\mathsf{ok}_m$ |
| $F; V \models_m \mathsf{false}$ | iff never |

**Fig. 5.** Assertion Semantics

### 4.4 Properties

Let fv($P$) for a proposition $P$ represent the free variables of $P$. Semantic entailment, $P \models_m Q$, holds between two formulae under the fault model $m$ iff for all $F$ and $V$ such that $\text{fv}(Q) \cup \text{fv}(P) \subseteq \mathit{dom}\ V$, $F; V \models_m Q$ whenever $F; V \models_m P$. The resulting logic has the following useful properties:

**Proposition 1.**

- *$*$ is commutative and associative with unit* $\mathsf{true}$.
- *If $P * Q$ holds, then so does $P$.*
- *$P \vee P$ is equivalent to $P$.*
- *If $P' \models_m P$ and $Q' \models_m Q$, then $P' * Q' \models_m P * Q$.*

- *In any state, if* $\forall\phi_1, \phi_2.\ \mathsf{hap}\,\phi_1 * \mathsf{hap}\,\phi_2 \mathrel{-\!\!*} P$ *holds, then so does* $\forall\phi_1.\ \mathsf{hap}\,\phi_1 \mathrel{-\!\!*} \forall\phi_2.\ \mathsf{hap}\,\phi_2 \mathrel{-\!\!*} P$.
- $\mathsf{faulty}\ f$, $\mathsf{id}\ f$, *and equality of expressions are independent of well-formed fault states.*
- *If* $F_1 + F_2\ \mathsf{ok}_m$ *and* $F_1, V \vDash_m P$, *then* $F_1 + F_2, V \vDash_m P$.

*Proof.* Immediate using the semantics of assertions.

**Lemma 1.** *For all assertions* $P$, *fault states* $F$, *environments* $V$, *variables* $x$, *and expressions* $E$, $F; V \vDash_m P[E/x]$ *iff* $F, V[x \mapsto E] \vDash_m P$.

*Proof.* By induction on structure of $P$, simultaneously for the if and only if directions. This is necessary to get the inductive hypothesis in both directions for the $\mathrel{-\!\!*}$ case.

**Proposition 2.** *The Hoare Logic fault rule, Hfault, is sound with respect to the assertion semantics.*

*Proof.* By induction on the derivation of $\{P\}\texttt{fault x}\{Q\}_m$. Uses the above substitution lemma for the fault rule case.

**Proposition 3.** *The fault rule generates the weakest precondition, in the strong sense that for any* $F$ *and* $V$ *that do not entail the precondition, and any* $F'$ *and* $V'$ *such that* $(F, V, \texttt{fault x}) \longmapsto (F', V', -)$, *it is the case that* $F'; V'$ *does not entail the postcondition.*

*Proof.* Easy proof from the definitions.

For every statment but the fault statement, the frame rule is standard. Here we verify that the frame rule holds for the fault statement as well.

**Proposition 4.** *The frame rule holds for the fault statement:*

$$\frac{\{P\}\texttt{fault x}\{Q\}_m}{\{P * R\}\texttt{fault x}\{Q * R\}_m}\, x \notin \mathrm{fv}(R)$$

*Proof.* By induction on the derivation of $\{P\}\texttt{fault x}\{Q\}_m$.

## 5 Taming Proof Complexity

The large number of fault functions generated by the fault rule can make it difficult to manage proofs in the program logic. Even quite simple programs can require manipulation and reasoning about many fault functions. For example, the program in Figure 6 redundantly computes a single addition three times and compares the results. Even such a simple program generates a large and unwieldy precondition that includes nine different universally quantified variables. Fortunately, though the apparent complexity grows quickly, the reasoning itself is relatively simple. In this section, we show how to tame such complexity by introducing a new modal operator.

$$\{a_0 = a \,\&\, a_1 = a \,\&\, a_2 = a \,\&\, b_0 = b \,\&\, b_1 = b \,\&\, b_2 = b\}_m$$

$\vdots$ (sequence of entailments elided)

$$\{\forall \phi_{a_0}, \phi_{b_0}.\ \forall \phi_{a_1}, \phi_{b_1}.\ \forall \phi_{a_2}, \phi_{b_2}.\ \forall \phi_0, \phi_1, \phi_2.\ \mathsf{hap}\,(\phi_{a_1}) * \mathsf{hap}\,(\phi_{b_1}) \\ *\mathsf{hap}\,(\phi_{a_2}) * \mathsf{hap}\,(\phi_{b_2}) * \mathsf{hap}\,(\phi_0) * \mathsf{hap}\,(\phi_1, \phi_2\}) \mathrel{-\!\!*} \\ (\phi_1(\phi_{a_1}a_1 + \phi_{b_1}b_1) = \phi_2(\phi_{a_2}a_2 + \phi_{b_2}b_2) \,\&\, \phi_1(\phi_{a_1}a_1 + \phi_{b_1}b_1) = a + b) \vee \\ (\phi_1(\phi_{a_1}a_1 + \phi_{b_1}b_1) \neq \phi_2(\phi_{a_2}a_2 + \phi_{b_2}b_2) \,\&\, \phi_0(\phi_{a_0}a_0 + \phi_{b_0}b_0) = a + b)\}_m$$

```
fault a0, b0;
a0 = a0 + b0;
fault a1, b1;
```

```
a1 = a1 + b1;
```

$\vdots$ (this is the complex part)

```
fault a2, b2;
a2 = a2 + b2;
fault a0, a1, a2;
```

$$\{(a_1 = a_2 \,\&\, a_1 = a + b) \vee (a_1 \neq a_2 \,\&\, a_0 = a + b)\}_m$$

```
if a1=a2
  then a0 = a1;
  else skip;
```

$\vdots$

$$\{a_0 = a + b\}_m$$

**Fig. 6.** An elided version of a complicated example with $m$ = SWC fault model

### 5.1 The Possibility Modality

To eliminate the need to deal with universally quantified fault functions directly, we have hidden them inside a modal operator $\bigcirc P$, read "maybe $P$" and meaning "$P$ is true in the absence of faults." More precisely, $\bigcirc P$ says that either $P$ is true, or a fault has occurred.

$$\bigcirc P \overset{\text{def}}{=} (\exists \phi.\ \mathsf{hap}\,\phi * \mathsf{faulty}\,\phi) \vee P$$

The key property of $\bigcirc$ is its relation to the fault statement in our Hoare Logic. The modality $\bigcirc$ allows for a simple Hoare rule, as `fault x` preserves $\bigcirc P$ for any $P$.

**Proposition 5.** *$\{\bigcirc P\}$`fault x`$\{\bigcirc P\}_m$ is valid for all $P$.*

*Proof.* This follows by proving that the precondition obtained by applying the Hfault rule to $\bigcirc P$ implies $\bigcirc P$. Uses substitution lemma 1.

By combining this Hoare rule with the frame rule for `fault x`, we obtained

$$\{\bigcirc P * Q\}\texttt{fault x}\{\bigcirc P * Q\}_m$$

whenever $x \notin \mathrm{fv}(Q)$. These $\bigcirc$-based Hoare rules for the `fault` statement do not contain any explicit fault functions, allowing us to ignore the fault functions in cases when the new rules apply.

Under the SWC fault model an additional and quite useful property holds:

**Proposition 6.** *Under the SEU fault model*

$$\bigcirc P \ast \bigcirc Q \vDash_m P \vee Q$$

*and, in a generalized form:*

$$\ast_{i=1}^{n} \bigcirc P_i \vDash_m \bigvee_{j=1}^{n} \&_{i=\{1,\ldots,n\}\setminus\{j\}} P_i$$

*Proof.* By case analysis on whether and where a fault occurs.

This enables the easy derivation of useful postconditions to programs using modular redundancy. Using this rule with the Hoare rule involving $\bigcirc$, we can derive postconditions such as those of the form $\langle$result is correct$\rangle \vee \langle$other result is correct$\rangle$ where the two results come from modular computations.

With $\bigcirc$, the rough example from Section 5 is much simpler, as seen in Figure 7. Though still relatively long, this proof is quite simple and regular. There is not a single visible quantifier or fault function in the proof. What was formerly the most complex part of the proof now only has one simple assertion per line of code.

```
                      ${a_0 = a \& a_1 = a \& a_2 = a \& b_0 = b \& b_1 = b \& b_2 = b}_m$
                      ${a_0 + b_0 = a + b \ast a_1 + b_1 = a + b \ast a_2 + b_2 = a + b}_m$
                      ${\bigcirc a_0 + b_0 = a + b \ast \bigcirc a_1 + b_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
fault $a_0$, $b_0$;          ${\bigcirc a_0 + b_0 = a + b \ast \bigcirc a_1 + b_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
$a_0$ = $a_0$ + $b_0$;       ${\bigcirc a_0 = a + b \ast \bigcirc a_1 + b_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
fault $a_1$, $b_1$;          ${\bigcirc a_0 = a + b \ast \bigcirc a_1 + b_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
$a_1$ = $a_1$ + $b_1$;       ${\bigcirc a_0 = a + b \ast \bigcirc a_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
fault $a_2$, $b_2$;          ${\bigcirc a_0 = a + b \ast \bigcirc a_1 = a + b \ast \bigcirc a_2 + b_2 = a + b}_m$
$a_2$ = $a_2$ + $b_2$;       ${\bigcirc a_0 = a + b \ast \bigcirc a_1 = a + b \ast \bigcirc a_2 = a + b}_m$
fault $a_0$, $a_1$, $a_2$;   ${\bigcirc a_0 = a + b \ast \bigcirc a_1 = a + b \ast \bigcirc a_2 = a + b}_m$
                      $P \overset{\text{def}}{=} {(a_1 = a_2 \& a_1 = a + b) \vee (a_1 \neq a_2 \& a_0 = a + b)}_m$
if $a_1$=$a_2$
                        ${a_1 = a_2 \& P}_m$
                        ${a_1 = a + b}_m$
  then $a_0$ = $a_1$;          ${a_0 = a + b}_m$
                        ${a_1 \neq a_2 \& P}_m$
                        ${a_0 = a + b}_m$
  else skip;            ${a_0 = a + b}_m$
                      ${a_0 = a + b}_m$
```

**Fig. 7.** The previous example, but smoother, $m$ = SWC fault model

## 6 RSA Sign/Verify

We now describe a more realistic example using the RSA Sign/Verify algorithm, one of many algorithms used to authenticate messages using digital signatures. RSA is a very widely used public key encryption system based on the difficulty of factoring a product of two large primes, $n = p \cdot q$. A public and private key, called $e$ and $d$, respectively, are generated such that $e \cdot d \equiv 1 \bmod ((p-1)*(q-1))$. When used for digital signatures, a signature is created by starting with a hash of the message and exponentiating it by raising it to the power given by the private key, modulo $p \cdot q$. The message and signature are then sent out. A recipient can *verify* the sender of the message by raising the signature to the power of the public key, modulo $p \cdot q$, and comparing this to the hash of the received message.

A common implementation of RSA uses the Chinese remainder theorem to speed up the exponentiation. The exponentiation is done twice, once modulo $p$ and once modulo $q$. Then the results are multiplied by precalculated constants and added together. The same number of multiplications must be calculated, but the numbers are half the length in bits, so each multiplication takes about a quarter of the time. Thus there is an overall speedup of about 4.

However, Boneh and DeMilo showed that a single fault during execution of the Chinese remainder theorem algorithm for RSA not only fails validation, but can also compromise the secret key. As such, it is important to protect the algorithm with appropriate redundancy. One way to do so is to use a calculate-and-check form of fault tolerance where the check is simply the verify portion of the RSA algorithm. The verify step is also particularly fast, as the exponent used to decrypt the signature, $e$, is chosen so that it has a short bit length (commonly $e$ is 65537, 17 bits long), enabling a very quick exponentiation. Using our system, we have proven the version of the RSA Sign/Verify algorithm appearing in Figure 8 fault tolerant with respect to the SWC Fault Model.

## 7 Certifying Compilation with Triple Modular Redundancy

In addition to being used as a standalone logic for proofs about fault tolerant programs, our logic can be used within the context of a certifying compiler to guarantee the compiler outputs fault tolerant code. To demonstrate this idea, we have developed a formal translation from ordinary, non-fault-tolerant Hoare triples, proven sound using conventional Hoare rules, into fault-tolerant Hoare triples proven sound with respect to the SWC fault model in our logic. The compiler achieves generic fault tolerance by adding triple modular redundancy to the program. In other words, each subexpression is recomputed three times and the results are compared to detect faults. Figure 9 presents the translation, which is composed of independent judgements for translating expressions ($B \rightsquigarrow B'$ for booleans and $E \rightsquigarrow (E_1, E_2, E_3)$ for integer expressions (there is one translated expression for each redundant computation)), statements ($S \rightsquigarrow S'$), and Hoare triples ($\{P\}\mathtt{S}\{Q\}_m \rightsquigarrow \{P'\}\mathtt{S'}\{Q'\}_m$). The top level translation of Hoare triples is performed according to the rule Ttriple, the program being translated according

$$\{(\forall x, c \in V : c^e = x(\text{mod } n) \rightarrow c = x^d(\text{mod } n)) * d < 2^{512} * e < 2^{17} * \mathtt{ev} = e * \mathtt{nv} = p \cdot q * \mathtt{s2} = 1 * \mathtt{i2} = 17 * \mathtt{mv2} = m * (\forall s_1, s_2, x. s_1 = x^{d_p}(\text{mod } p) * s_2 = x^{d_q}(\text{mod } q) \mathrel{-\!\!*} a \cdot s_1 + b \cdot s_2 = x^d(\text{mod } p \cdot q)) * \mathtt{av} = a * \mathtt{bv} = b * \mathtt{dvq} = d_q * \mathtt{qv1} = q * \mathtt{mvq} = m * d < 2^{17} * \mathtt{sp} = 1 * \mathtt{dvp} = d_p * \mathtt{pv1} = p * \mathtt{mvp} = m * \mathtt{ip} := 511\}$$

*Calculate signature modulo p.*

```
fault ip
while ip > -1
  fault sp, pv1
  sp := sp*sp (mod pv1)
  fault dvp, ip
  if dvp & (1 << ip) != 0:
    fault sp, mv, pv1
    sp := sp * mv (mod pv1)
  else:
    skip
  fault ip
  ip--
  fault ip
```

*Calculate signature modulo q.*

```
fault iq
while iq > -1
  fault sq, qv1
  sq := sq*sq (mod qv1)
  fault dvq, iq
  if dvq & (1 << iq) != 0:
    fault sq, mvq, qv1
    sq := sq * mvq (mod qv1)
  else:
    skip
  fault iq
  iq--
  fault iq
```

*Combine results to get actual signature.*

```
fault sp, av
tp := sp * av
fault sq, bv
tq := sq * bv
fault tp, tq
s := tp + tq
```

*Check for errors by performing verify.*

```
good := 1
fault s
out := s
fault i2
while i2 > -1:
  fault s2, nv2
  s2 := s2*s2 (mod nv2)
  fault ev2, i2
  fault ev2, i2
  if ev & (1<<i2) != 0:
    fault s2, out, nv2
    s2 := s2 * out (mod nv2)
  else:
    skip
  fault i2
  i2--
  fault i2 fault mv2, s2
if mv2 != s2:
  good := 0
else:
  skip
```

$$\{\mathsf{good} = 0 \vee \mathsf{s} = m^d(\text{mod } n)\}$$

**Fig. 8.** RSA Message Signing with Chinese Remainder Theorem, Fault Tolerant, SWC Fault Model

Translation of Boolean and Integer Expressions:

$$\text{Tbool}\ \frac{}{B \rightsquigarrow \text{majority-vote}\,(B_1, B_2, B_3)}\ \text{where } B_i \text{ is } B \text{ with an } i \text{ subscript added to each variable name.}$$

$$\text{Texpr}\ \frac{}{E \rightsquigarrow (E_1, E_2, E_3)}\ \text{where } E_i \text{ is } E \text{ with an } i \text{ subscript added to each variable name.}$$

Translation of Imperative Statements:

$$\text{Twhile}\ \frac{B \rightsquigarrow B' \qquad S \rightsquigarrow S'}{\texttt{while}\ B\ \texttt{do}\ S \rightsquigarrow \texttt{fault}\ fv(B')\ \texttt{;}\ \texttt{while}\ B'\ \texttt{do}\ \texttt{(}S'\ \texttt{;}\ \texttt{fault fv(}B'\texttt{))}}$$

$$\text{Tseq}\ \frac{S \rightsquigarrow S' \qquad T \rightsquigarrow T'}{S\ \texttt{;}\ T \rightsquigarrow S'\ \texttt{;}\ T'}$$

$$\text{Tif}\ \frac{B \rightsquigarrow B' \qquad S \rightsquigarrow S' \qquad T \rightsquigarrow T'}{\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T \rightsquigarrow \texttt{fault}\ fv(B')\ \texttt{;}\ \texttt{if}\ B'\ \texttt{then}\ S'\ \texttt{else}\ T'}$$

$$\text{Tskip}\ \frac{}{\texttt{skip} \rightsquigarrow \texttt{skip}}$$

$$\text{Tasgn}\ \frac{E \rightsquigarrow (E_1, E_2, E_3)}{x\ \texttt{:=}\ E \rightsquigarrow \texttt{fault}\ fv(E_1)\ \texttt{;}\ x_1\ \texttt{:=}\ E_1\ \texttt{;}\ \texttt{fault}\ fv(E_2)\ \texttt{;}\ x_2\ \texttt{:=}\ E_2\ \texttt{;}\ \texttt{fault}\ fv(E_3)\ \texttt{;}\ x_3\ \texttt{:=}\ E_3}$$

Translation of Hoare triples:

Let $\text{convert}[P] \overset{\text{def}}{=} \exists \mathbf{x}'.\ \bigcirc(\mathbf{x}_1 = \mathbf{x}') * \bigcirc(\mathbf{x}_2 = \mathbf{x}') * \bigcirc(\mathbf{x}_3 = \mathbf{x}') * P[\mathbf{x}'/\mathbf{x}]$ where $\mathbf{x}$ is the vector of program variables in $P$.

$$\text{Ttriple}\ \frac{S \rightsquigarrow S'}{\{P\}S\{Q\} \rightsquigarrow \{\text{convert}[P]\}S'\{\text{convert}[Q]\}_m}$$

**Fig. 9.** Translation from Program and Specification in standard Hoare logic to Triple Modular Redundant Program in our logic

to the rules for translating statements and the precondition and postcondition being converted by the convert predicate.

The most interesting aspect of the translation is the coding of triple modular redundancy in our assertion logic: Given a standard assertion $P(x)$, which refers to some (non-fault-tolerant) program variable $x$, the translated assertion will have the form $\exists \mathbf{x}'.\ \bigcirc(\mathbf{x}_1 = \mathbf{x}') * \bigcirc(\mathbf{x}_2 = \mathbf{x}') * \bigcirc(\mathbf{x}_3 = \mathbf{x}') * P[\mathbf{x}'/\mathbf{x}]$. Intuitively, this assertion states that states that $P(\mathbf{x}')$ will be true and $\mathbf{x}'$ may be equal to any one of three redundant versions of the original variable $x$, called $x_1$, $x_2$, and $x_3$. Additionally, when working in the SWC fault model, at most one of $x_1$, $x_2$, or $x_3$ will not be equal to $\mathbf{x}'$, allowing us to conclude at least two of the three assertions $P(x_1)$, $P(x_2)$ and $P(x_3)$ are true. By comparing $x_1$, $x_2$, and $x_3$ to each other, one can determine which (if any) variables are faulty and hence which predicates are true.

**Proposition 7.** *Given a valid standard Hoare triple as input, the translation produces a valid logic Hoare triple in our logic as output.*

## 8 Related Work

There are many existing methods for mitigating the effects of transient faults, using both hardware mechanisms, software mechanisms, and combinations of the two. For example, many solutions in software [10–13] require the compiler to duplicate computations and to insert comparisons to ensure that the two copies remain in agreement. Such techniques are usually evaluated experimentally using random fault injection, which shows that these solutions handle large classes of faults, but gives no hard and fast semantic guarantees about program behavior.

The SymPLFIED system [14] is a notable exception to the practice of random fault injection. SymPLFIED uses model checking to iterate through all possible hardware faults and to determine whether such faults can lead to catastrophic outcomes in the application being analyzed. SymPLIFIED has a significantly richer error model than the ones treated in this paper as it considers memory errors and control-flow errors. On the other hand, SymPLIFIED does not come with a program logic, like the one defined in this paper, that makes it possible to judge whether a program satisfies some general-purpose logical specification.

Another closely related line of research involves the development of type systems for checking fault tolerance properties. For example, the faulty lambda calculus, $\lambda_{zap}$ [15], uses a type system to ensure its programs use triple modular redundancy properly. Elsman [16] shows how to extend that calculus with simplified error detection operations. More recent work applies these abstract, high-level ideas directly to assembly langauge [17, 18]. The main drawback of these type-based approaches is that each new fault tolerance scheme requires its own type system. In contrast, this paper proposes a more general logical framework for understanding how transient faults affect software behavior.

## 9 Conclusion

While development of most applications does not require reasoning about transient hardware faults, there are several domains in which such faults can cause substantial problems. One domain of particular interest is in the development of cryptographic algorithms where recent research has shown that even a single fault induced by an attacker is often sufficient to break the security of well-known algorithms such as RSA and DES.

This paper makes initial progress in the development of a framework for verifying such programs. It shows how to extend the operational semantics of a simple language of while programs with standard fault models and develops a variation of Separation Logic to reason about these programs and their faults. It also shows how to define and use a modal operator to simplify certain proofs of fault tolerance. Finally, the paper presents two illustrative applications of the logic: one involving a fault tolerant version of RSA and a second involving a compiler transformation that introduces triple modular redundancy.

*Acknowledgments.* The authors would like to acknowledge the help of Frances Perry in the early stages of the work presented in this paper. This research is funded in part by NSF award CNS-0627650. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

## References

1. Boneh, D., DeMillo, R., Lipton, R.: On the importance of checking cryptographic protocols for faults. Journal of Cryptology **14**(2) (2001) 101–119
2. Govindavajhala, S., Appel, A.: Using memory errors to attack a virtual machine. In: Proceedings of the 2003 Symposium on Security and Privacy. (May 2003) 153–165
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE **94**(2) (Feb. 2006) 370–382
4. Hegde, R., Shanbhag, N.R.: Energy-efficient signal processing via algorithmic noise-tolerance. In: ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design, New York, NY, USA, ACM (1999) 30–35
5. Chang, J., Reis, G.A., August, D.I.: Automatic instruction-level software-only recovery methods. In: Proceedings of the 2006 International Conference on Dependendable Systems and Networks. (June 2006)
6. Shirvani, P.P., Saxena, N., McCluskey, E.J.: Software-implemented EDAC protection against SEUs. In: IEEE Transactions on Reliability. Volume 49. (2000) 273–284
7. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1996)
8. Ishtiaq, S., O'Hearn, P.: Bi as an assertion language for mutable data structures. In: Proceedings of the 28th ACM Symposium on Principles of Programming Languages, London, United Kingdom (January 2001) 14–26
9. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society (2002) 55–74
10. Borin, E., Wang, C., Wu, Y., Araujo, G.: Software-based transparent and comprehensive control-flow error detection. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 333–345
11. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. **51**(2) (March 2002) 111–122
12. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: Proceedings of the 3rd International Symposium on Code Generation and Optimization. (March 2005)
13. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Design and evaluation of hybrid fault-detection systems. In: Proceedings of the 32th Annual International Symposium on Computer Architecture. (June 2005) 148–159
14. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: Symplfied: Symbolic program-level fault injection and error detection framework. In: International Conference on Dependable Systems and Networks. (2008)
15. Walker, D., Mackey, L., Ligatti, J., Reis, G., August, D.I.: Static typing for a faulty lambda calculus. In: ACM International Conference on Functional Programming, Portland, Oregon (September 2006)
16. Elsman, M.: 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)
17. Perry, F., Mackey, L., Reis, G.A., Ligatti, J., August, D.I., Walker, D.: Fault-tolerant typed assembly language. In: International Symposium on Programming Language Design and Implementation (PLDI). (June 2007)
18. Perry, F., Walker, D.: Reasoning about control flow in the presence of transient faults. In: International Static Analysis Symposium. (July 2008)