TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik, Institut für Systemarchitektur

TECHNISCHE BERICHTE
TECHNICAL REPORTS

ISSN 1430-211X

TUD-FI15-01-Februar 2015

Diogo Behrens[1], Marco Serafini[2], Sergei Arnautov[1],
Flavio P. Junqueira[3], Christof Fetzer[1]

[1] Technische Universität Dresden, Germany
[2] Qatar Computing Research Institute, Qatar
[3] Microsoft Research Cambridge, UK

Scalable error isolation for distributed systems: modeling, correctness proofs, and additional experiments

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
01062 Dresden, Germany

DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

Scalable error isolation for distributed systems: modeling, correctness proofs, and additional experiments

Diogo Behrens[*], Marco Serafini[◇], Sergei Arnautov[*], Flavio P. Junqueira[‡], Christof Fetzer[*]
[*]Technische Universität Dresden, Germany
[◇]Qatar Computing Research Institute, Qatar
[‡]Microsoft Research, Cambridge, UK

Contents

1 Introduction

This technical report complements our paper entitled "Scalable error isolation for distributed systems" [3].

In Section 2, we define the distributed system model and formalize the central property of this work: error isolation. The property dictates that correct processes discard any corrupt input message. Error isolation can be achieved with traditional techniques such as Byzantine-fault tolerant algorithms. In this work, however, we are interested in techniques that provide error isolation locally, *i.e.*, exchanging no additional messages among processes. In Section 2.4.2, we define *hardening* as a transformation of a program p into a program p_h, such that if all processes of the system run such program p_h, then error isolation is guaranteed locally. If our system is only subject to hardware errors, but no malicious attacks, hardening can guarantee error isolation with broad coverage.

Section 3 presents the fault model upon which hardening is constructed. Our fault model describes the effect and extent of arbitrary faults *at the process level* and is based on the ASC fault model by Correia *et al.* [9]. In contrast to their ASC fault model, however, the model presented in this work is more general, being not tied to a specific hardening approach.

In Section 4, we describe the Scalable Error Isolation (SEI) technique in detail, including model refinements, fault assumptions, and a correctness proof for single-thread programs. In Section 5, we extend the definitions, algorithms, and proof to support hardening of multithreaded programs.

Finally, in Section 6, we present additional experimental results and details on the setup used in the experiments of our companion paper [3].

2 Error isolation in distributed systems

We start by introducing the system model and defining several terms that are used throughout this work. This section motivates and sets the basis for the formalism we present starting in Section 3.

2.1 System model

A *distributed system* is a set of processes $\Pi = \{\pi_1, \ldots, \pi_n\}$, with $n > 1$, that communicate via message-passing over a network. The processes and the network are the *components* of the distributed system. We use in our definitions the time of a wall clock not accessible by the system.

Each component of the system is modeled as a state machine, which consists of *state variables* and *state transitions*. A *state* is an assignment of values to the state variables. The *system state* – also called *configuration* in the literature [14, 15] – encompasses the state of all system components, *i.e.*, processes and network. An *execution* – also called behavior [21, 20] – is *any* infinite sequence of system states. A *system execution* is an execution given by interleaved state transitions of the system components, starting from some initial system state. Finally, a *property* is defined as a set of executions [31].

There are two types of components in a system. The *network component* is simply a set into which and from which processes send and receive messages. The *processes* are precisely defined in Section 3.1. In a nutshell, a process π executes the following state transitions in a loop. First, π receives a message from the network, removing it from the network set and storing it in its input buffer. Next, π handles the input message by performing some computation based on its state and the message, modifying its internal state, and (possibly) generating one or more output messages, which are placed in π's output buffer. Finally, π sends any output message to the network, by removing the message from the output buffer and writing it into the network set. Note that the details of message receive and send are not relevant for our work; hence, we model them as single state transitions. The handling of a message is, however, represented as a series of state transitions.

2.2 Faults, errors, and failures

Intuitively, faults are defects or adversarial conditions that when activated can cause errors. Errors may propagate and become externally visible as failures.

In more detail, *faults*, also called *fault transitions* or *fault steps*, are special state transitions added to the components of the system [11, 21]. When a fault transition is taken in an execution, an error occurs. *Errors* are abnormal system states. A component commits a *failure* if an error becomes visible to other components of the system or to an external user. In other words, a failure is a deviation from the component's expected external behavior.

A component failure can manifest in different ways. A commission [19] or value failure [28] is an unexpected message or a message with unexpected content sent out by a component. An omission failure is the absence of a message expected to be sent by a component. More precisely, based on the current system state, an omniscient observer would expect a message being sent by the component, but none is sent. In system models that assume some form of synchrony, failures can also manifest on the time domain, being called timing failures [28] or performance failures [12]. Since we make no synchrony assumptions, we do not consider timing/performance failures.

2.3 Fault models

To conclude this section, we introduce two fault models: the crash-stop fault model and the arbitrary-fault model. They serve as baselines for our new fault model in Section 3.

2.3.1 Crash-stop fault model

In the crash-stop fault model [9], processes might crash and the network might lose, duplicate, reorder, or misroute messages. The most prevalent of these failures is the process crash. A process π commits a crash failure by performing a *crash-fault transition* into a halt state from which no further transition of π is possible. From the halt state, process π can only commit omission failures, sending no further message out. The remainder failures are due to the network component. Message omission, duplication, and reordering are well-known problems in point-to-point communication, and are typically solved by retrying and piggybacking sequence numbers to messages [4]. Misrouting can also be easily detected if the sender appends the process identifier of the destination process to each message it sends; and the receiver checks whether it is indeed the right destination process of every message it receives.

In general, algorithms designed to tolerate crash-stop failures are easier to reason about [7, 25] and in many cases preferable to arbitrary-fault-tolerant algorithms [10, 28]. In this work, we assume all distributed algorithms in consideration can tolerate crash-stop failures.

2.3.2 Arbitrary-fault model

The arbitrary-fault model extends components with *arbitrary faults*, which are fault transitions to any representable state of a component, *e.g.*, corruption of messages in the network, corruption of a variable in a process, substitution of the entire operating system image, etc. Besides failing with crash-stop modes, a component might fail arbitrarily by committing a value failure, *i.e.*, by sending *corrupt messages* out. We define a corrupt message as follows.

A generation history for a message m is a sequence of messages that could be received by a correct process in order to generate m.

Definition 1 (Message precedence) *Let π be a process. Let m_{out} and m'_{out} two output messages. Let m'_{in} be the input message π handled in order to produce m'_{out}. We say that m_{out} precedes m'_{out} if and only if π sends m_{out} before it receives m'_{in}.*

A process π receives the input message m when it starts executing the event handler that takes m as input.

Definition 2 (Generation history of a message) *Let π be a faulty process. Let h be a subsequence of the sequence of correct messages received by π. Let m be a message sent by π. The subsequence h is a generation history for m if there exists a run in which π is correct and π outputs m after receiving each message in h.*

Note that a message can have multiple generation histories, corresponding to multiple runs where the message might have been produced.

A correct message m is one that has a *correct* generation history h, that is, a generation history guaranteeing that previous history is not lost. The inductive definition is as follows.

Definition 3 (Correct generation history of a message) *Let π be a faulty single-threaded process. Let m be a message sent by π.*

- *If π has sent no message m' before m such that m' has a correct generation history, then all generation histories of m are correct for m.*
- *Else, for each output message m' preceding m, let H be set of correct generation histories of m'. A generation history of m is correct if and only if it extends some generation history in H.*

This definition refers to a single-threaded process where the precedence relationship among messages is a total order. We will consider the multithreaded case in Section 5.

Definition 4 (Correct message) *Let π be a faulty process. A message m currently sent by π is correct if and only if it has a correct generation history.*

Definition 5 (Corrupt message) *A message m is corrupt if and only if it is not correct.*

This definition of correct message forces every correct output message to take into consideration the effect of all correct output messages that have been processed previously and became externally visible. Messages received together can be processed in any order, but they must result in consistent histories.

Messages are always part of the state of some component. Hence, a corrupt message can be equivalently defined by referring to the state representing the message inside the component. If that state is corrupt, the message is corrupt; and vice versa. Such a definition of corrupt message, however, requires a precise notion of state corruption, which we first introduce in Section 3.

Beyond corrupt messages, arbitrary faults can manifest as omission failures [19] as well, but we attribute omission failures to the network.

Remarks on the Byzantine-fault model. From a failure perspective, the arbitrary-fault model and the Byzantine-fault model [22] are equivalent. Nevertheless, Byzantine faults often only model the specification deviation, *i.e.*, the value or omission failure, ignoring the actual internal state corruption – see, for example, the definition by Castro and Liskov [5]. That is not a problem, but a feature of the Byzantine-fault model. One can abstract away the causes of a Byzantine failure and simply model the Byzantine failures as fault transitions.

In this work, however, we are interested precisely in the arbitrary faults that become Byzantine failures (*i.e.*, arbitrary failures). We model an arbitrary-fault transition as an arbitrary change of the

Classification	**Failure**	**Fault**
correct	no	–
crashed	crash	crash transition
faulty	no	arbitrary transition
contaminated	no	corrupt message received and processed
failed	corrupt message	from faulty or contaminated

Table 1: Process classification according to failure and fault

internal state of a system component. (Normal) state transitions might then propagate the caused error to finally become a Byzantine failure. This level of modeling is particularly important to reason about the correctness of SEI because, as we will define in the next section, the challenge in devising a hardening technique such as SEI is in detecting internal state corruptions before they become arbitrary failures.

2.4 Problem definition

In an execution subject to arbitrary faults, an omniscient observer can classify a process π as correct, crashed, faulty, contaminated, or failed. Table 1 relates this classification with the faults and failures process π suffers and commits. In this work, it is sufficient to classify only the processes; we do not classify the network component as faulty, crashed, etc.

Process π is *correct* if it neither commits a failure nor suffers a fault. Process π becomes *crashed* if it fails by performing a crash transition. If process π fails by sending a corrupt message, then π is classified as *failed*, independently of the fault that caused the failure. If process π suffers a fault other than a crash transition, π does not have to immediately fail; in fact, it may never fail. While process π is in an erroneous state, but not yet failed, p is classified as faulty or contaminated depending on which type of fault π has suffered first. A process π becomes *faulty* if π performs an arbitrary fault transition. A process π becomes *contaminated* if π receives, processes, and modifies its state according to a corrupt message sent by some other component – the other component being the network or another process.

2.4.1 Error isolation

The central objective of this work is coping with failure propagation due to arbitrary faults in distributed systems. We call *failure propagation* the contamination of a process as a result of an arbitrary failure of another component. Different from faulty processes, contaminated processes are induced to transition into an erroneous state by an external fault source. One cannot restrict by assumption how many processes might get contaminated in this way because a contaminated process can again contaminate other correct processes, disrupting the whole system at some point as in Amazon S3 case [13].

Processes of a fault-tolerant system do not propagate failures as long as the system guarantees *error isolation*, which is defined as follows.

Property 1 (Error isolation) *A correct process π discards a received message m without modifying its state according to m if m is corrupt.*

A corollary of Property 1 is that, if error isolation holds, no correct process is ever contaminated.

Error isolation could be easily achieved by crashing correct processes. To rule out such solutions, we also need this additional property.

Property 2 (Accuracy) *A correct process π is never induced to crash or discard a correct message.*

2.4.2 Hardening

Under the crash-stop model, error isolation is trivially guaranteed. A process follows its specification until it crashes, sending no corrupt messages. In the arbitrary-fault model, error isolation is challenging to guarantee. Byzantine-fault tolerance is one way to achieve error isolation at the cost of additional hardware components [5, 6, 23]. In this work, we exclusively focus on faults caused by transient hardware errors as opposed to malicious adversaries or bugs. We argue that *hardening* can make the probability of failure propagation negligible.

To define hardening precisely, we first introduce the concept of message validity. This definition of message validity is based on the observation that often error-detection codes, *e.g.*, cyclic redundancy check (CRC), are used to protect messages during transmission [32]. The error-detection codes define a syntax of messages that can be accepted as correct.

Definition 6 (Message validity) *Let M be the set of all possible messages sent in the system. Let $CV \subset M$ define the set of messages that pass the acceptance test of a given error-detection code. A message m is valid iff $m \in CV$, otherwise m is invalid.*

Note that invalid messages and corrupt messages are not the same. Valid and invalid messages are statically defined for all executions with the set $CV \subset M$ – any valid message m is contained in CV, whereas any invalid message m is contained in $M \setminus CV$. Whether a message is correct or corrupt, however, depends on the sequence of messages received by a process (see Definition 5). If no fault occurs, a correctly-implemented error-detection code never classifies correct messages to be invalid.

Any corrupt but valid message is a potential threat to the system if the system is not designed to cope with arbitrary failures. The goal of hardening is to make corrupt but valid messages impossible in any execution of the system. In a hardened system, a correct process can determine if a message is correct by directly inspecting the error-detection code piggybacked in the message.

Hardware errors in the network. Using the concept of message validity, the following assumption rules out the arbitrary behavior of the network component that is not caused by hardware errors.

Assumption 1 (No spurious messages) *If a valid message m is received by a process π_r at a time t_r, then another process π_s sent m at time $t_s \leq t_r$.*

In other words, Assumption 1 asserts that the network never creates valid messages as if it were a process, except for duplicated messages. If the network component does create a message, then the message is invalid.

Assumption 1 has a low assumption coverage in systems subject to malicious adversaries. A hacker could break the error-detection code and insert corrupt but valid messages in the compromised network component. However, if the system is never subject to attacks, which is our focus in this work, arbitrary faults have a negligible probability of producing a valid message. A message encompasses header fields, checksum, payload, etc. Creating an corrupt but valid message spontaneously would require one or more highly improbable fault transitions in the network component. Note that valid messages might still be duplicated, but duplicates are tolerated since they are in the set of crash-stop failures.

Hardware errors in the processes. In the hardening problem, message validity should be *more than a checksum* protecting the message against corruption of the network; it should represent an *evidence of good behavior* of the process sending the message. Intuitively, *hardening* is any software-only, process-local technique that translates "good" and "bad" behavior of a process into the validity of the messages. By *software-only* we mean a technique that requires no additional hardware components or processes to work – although it might use additional hardware to minimize its overhead. By *process-local* we mean a technique that requires no additional communication between processes, *i.e.*, the process alone decides whether it is behaving correctly or not and translates this information as message validity.

In an environment with no spurious messages (Assumption 1), a hardening technique should essentially achieve the following two properties in addition to accuracy (Property 2) to guarantee error isolation.

Property 3 (Local error exposure) *For any output message m of a faulty process π, if m is corrupt, then m is invalid,* i.e.*, $m \notin CV$.*

Property 4 (Local error filtering) *For any message m received by a correct process π, if $m \notin CV$, then π discards m without changing its state.*

Together these properties define the *hardening* problem.

Definition 7 (Hardening) *A hardening technique transforms a native program p into a hardened program p_h such that a process π executing program p_h guarantees local error exposure (Property 3), local error filtering (Property 4), and accuracy (Property 2).*

A *hardened process* is a process executing a hardened program. Definition 7 asserts that a correct hardened process never changes its state according to any invalid message received. More importantly, a faulty hardened process never sends out any corrupt but valid message.

Note that Property 3 constrains the behavior of *faulty* processes. That is, however, impossible under arbitrary faults. Independent of how the hardening is implemented, the information representing the detection of an error has to be present in some state variable or a combination of state variables. One or multiple arbitrary faults could always erase this information, leaving no traces back. Any hardening technique, consequently, has to make further assumptions about the format and/or frequency of the arbitrary faults. In Section 4.1, we formulate these additional assumptions for our SEI-hardening technique; in our paper [3] and in Section 6.1, we experimentally evaluate the coverage SEI's assumptions. For the sake of argument, we assume here such a hardening technique (*cf.* Definition 7) exists.

The following theorem directly results from Assumption 1 and Definition 7.

Theorem 1 *Let p be the native program running in a process $\pi \in \Pi$ of a given distributed system. If every process $\pi \in \Pi$ executes a hardened program p_h, then error isolation (Property 1) holds.*

Proof:
1. Every corrupt message is invalid.
   1.1. Any corrupt message sent out by the network is invalid by Assumption 1.
   1.2. Any corrupt message sent out by a hardened process $\pi \in \Pi$ is invalid by Property 3.
2. Error isolation (Property 1) holds.
   2.1. Correct processes discard any invalid messages by Property 4.
   2.2. Correct processes discard any corrupt messages by Steps 1 and 2.1.

□

3 Modeling process faults

In this section, we precisely model arbitrary process faults and build the framework upon which the SEI-hardening technique is constructed and its fault assumptions can be stated. Since our main goal is to prevent error propagation with techniques local to processes, we have to model the *faults* occurring inside components, in particular, inside processes – arbitrary faults occurring in the network are ruled out by Assumption 1.

We start by modeling the process execution, assuming processes work in a strict receive-handle-send loop (Section 3.1). Next, we present our arbitrary state corruption (ASC) fault model (Section 3.2). Since the fault model depends on the definition of the process model, we often refer to both models together simply as the ASC model.

3.1 Process model

A process π is a deterministic state machine composed of state variables and a set of state transitions.

3.1.1 Process state

We model the state s of a process as an assignment of values from a domain D to the set of variables V; the variables V represent memory locations, registers and the program counter. Note that the domain D is the same for all variables.

Definition 8 (State) *Given a set of variables V used by a process π and a values domain D, a state s of process π is a surjective function $s : V \rightarrow D$.*

We use the notation v to indicate a variable and $s[v]$ to indicate the value of the variable at state s – we loosely follow the TLA+ language notation [20], in which functions are denoted with square brackets. Unless noted otherwise, $s[v]$ represents the current value of variable v, whereas $s'[v]$ represents the value of v after the next state transition is performed. For simplicity, we occasionally indicate v as a value when the state including v is clear from the context. In particular, we indicate pc as the value of the program counter (*i.e.*, $s[pc]$), and pc' the next value of the program counter (*i.e.*, $s'[pc]$).

Program variables: A program p running on a process π does not have to use all variables in V (see Definition 12 for the definition of program p). We define the program variables $V_p \subset V$ to be the set of all variables potentially used by any execution of process π running a program p. We assume that V_p is a strict subset of all variables V, *i.e.*, there are always variables that are not used by p. This restriction facilitates the design of hardening mechanisms because it allows us to reason about all variables of a process, including the hardening-specific variables, before actually introducing the hardening technique.

Message buffer variables: Messages are represented with sets of variables in the state of processes. The set of variables $V_i \subset V_p$ represents the next input message to be handled, whereas the set of variables $V_o \subset V_p$ represents the next output messages to be sent. For convenience, we assume that $V_i \cap V_o = \{ \}$, *i.e.*, a process possesses an input buffer *and* an output buffer.

3.1.2 Handler programs and handle steps

The execution of a process π is a loop divided into three phases: message receipt, message processing, and message sending. These phases are three types of steps the process can take – note that a process may also take stuttering steps. To model the message processing phase, we need the concepts of instructions, operations and programs.

Definition 9 (Instruction) *An instruction i is a tuple $\langle operation, operands\rangle$.*

Definition 10 (Operation and operands) *An operation is a machine operation as understood from ordinary computers,* e.g.*, an addition, a subtraction, a conditional jump. The operands of an operation are specific variables or constant values used in the operation.*

Although there is a difference between instruction and operation, *i.e.*, an instruction is a concrete assignment of operands to an operation, we often use these terms interchangeably.

Operations can be branching and non-branching. The non-branching operations, *e.g.*, arithmetic operations, always increment the *pc* by 1, letting the program counter point to the following instruction. The branching operations set the *pc* according to their operands, *e.g.*, an unconditional branch $Jmp(v)$ sets the *pc* to the address in variable v.

Definition 11 (Source and target operands) *The operands of an instruction are divided in groups:* source operands *and* target operands*. An operation might use the value of the source operands, perform a computation and write the result in the target operands.*

Although the program counter *pc* is modified by all operations, *pc* is not part of the target operands of any instruction except when explicitly used as an operand. Therefore, branching instructions have no target operand since they only modify the *pc*.

We now define what programs are and how they are executed.

Definition 12 (Handler program) *A handler program p is an indexable sequence of instructions, denoted as $\langle i_1, \ldots, i_N\rangle$; in other words, a function $p : \{1, \ldots, N\} \to I$, mapping indices from 1 to N into the set I of all instructions.*

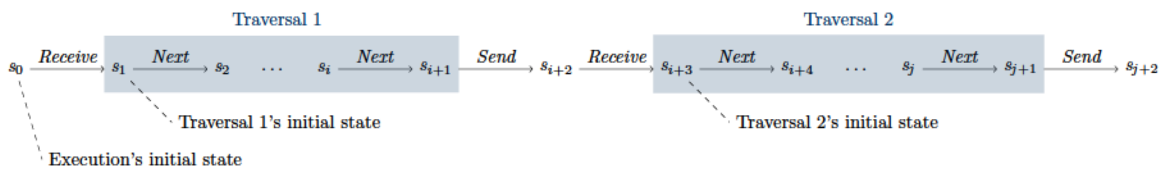
Definition 13 (Handle step) *Given a handler program p for process π, a handle step $Handle(s)$ applies the instruction $p[pc]$ to the current state s resulting in a new state s'.*

We allow only *well-formed* handler programs. Any well-formed handler program eventually terminates.

Definition 14 (Well-formed handler) *A handler program p is well-formed iff, for any correct state s with $pc = 1$, by successively applying handle steps, eventually an instruction of p sets $pc = N + 1$.*

By setting *pc* to $N + 1$, we model the termination of the handler and the return of control to the caller.

In our model, the instructions of a program p are not part of the variables V of the process. This modeling decision does not cause major limitations to our model, but simplifies its presentation. We discuss this point further in Section 3.2.



Traversal 1
Traversal 2
$s_0 \xrightarrow{Receive} s_1 \xrightarrow{Next} s_2 \cdots s_i \xrightarrow{Next} s_{i+1} \xrightarrow{Send} s_{i+2} \xrightarrow{Receive} s_{i+3} \xrightarrow{Next} s_{i+4} \cdots s_j \xrightarrow{Next} s_{j+1} \xrightarrow{Send} s_{j+2}$
Traversal 1's initial state
Traversal 2's initial state
Execution's initial state

Figure 1: Example of two traversals in some execution $E|_\pi$.

3.1.3 Process communication

Since process π is part of a distributed system, we also need to model communication. In addition to handle steps, a process can be modeled with a message receipt and a message sending step. A receive step reads a message m from the network and writes m into the predefined set of variables V_i. If there are messages to be sent out, a send step writes into the network the messages formed by the predefined set of variables V_o. Otherwise, the send step does nothing.

A process state transition can be either the receipt of a message if $pc = 0$, a program step $Handle(s)$ if $0 < pc \leq N$, or the sending of output messages if $pc = N + 1$. Formally, the following predicate holds for every state s of process π running a program p:

$$\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor\, pc = 0 \implies \mathit{Receive} \land pc' = pc + 1 \\
&\quad \lor\, pc > 0 \land pc \leq N \implies s' = \mathit{Handle}(s) \\
&\quad \lor\, pc = N + 1 \implies \mathit{Send} \land pc' = 0
\end{aligned}$$

Such a modeling, nevertheless, requires a precise definition of the receive and send steps. In turn, that further requires us to consider the whole distributed system to determine which messages can be sent in every step. Instead, we opt for a simpler, but equivalent, modeling using the concept of a traversal.

3.1.4 Traversals

A traversal is the part of an execution of a process π from the time when an input message is stored into the variables V_i – *i.e.*, the traversal's initial state – until the time the output messages in variables V_o are ready to be sent into the network – *i.e.*, the traversal's final state. Stated differently, a traversal of a process π running a program p is an execution of the handle steps while $1 \leq pc \leq N$. The execution of a process consequently is as a sequence of traversals interleaved with receive and send steps (see Figure 1 for an example execution with two traversals). With this modeling, we can reason about the correctness of traversals, instead of complete process executions.

A process state transition is defined as follows.

Definition 15 (Process state transition) *Given a program p for process π, a state transition Next is either a program step $Handle(s)$ if $1 \leq pc \leq N$ or stuttering steps once $pc = N + 1$. Formally, the following predicate holds for every state s:*

$$\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor\, pc \geq 1 \land pc \leq N \implies s' = \mathit{Handle}(s) \\
&\quad \lor\, pc = N + 1 \implies pc' = N + 1
\end{aligned}$$

The communication is modeled by many possible traversal initial states, each of them with a (potentially different) message stored in the variables V_i. Let I be such a set of traversal initial states. We now define a traversal.

Definition 16 (Traversal) *A traversal of process π running a program p is an execution starting from a traversal initial state $s \in I$ followed by states satisfying the process state transition Next or a stuttering step. Formally,*

$$\mathit{Traversal} \triangleq (s \in I) \land \Box(\mathit{Next} \lor s' = s)$$

The $\Box$ operator indicates that, for every pair of consecutive states $\langle s, s' \rangle$ in any traversal, the *Next* formula holds or the process stutters. The stuttering steps $s' = s$ are steps in which the process does nothing.

The difficulty of modeling traversals is exactly on defining the set of possible initial states I. For that, we use the set of all possible executions (of the complete system), and then select those states of π immediately after a message has been received by π, *i.e.*, immediately after a message has been written into the variables V_i and $pc = 1$. Figure 1 shows an example of two traversals in some execution of π. The traversal initial states are the first states within the traversals, *i.e.*, $\{s_1, s_{i+3}\} \subseteq I$.

Definition 17 (Traversal initial states I) *Let E be all possible executions of the system. Remember that an execution $e = \langle S_1, S_2, \ldots \rangle$ is a sequence of states $S_i = \langle s_{\pi_1}, \ldots, s_{\pi_N}, s_{network} \rangle$ comprising the state of processes in Π and network component. Let $E|_\pi$ be the set of executions E restricted to the states of process π. Finally, let A be the set of all states in all executions of π,* i.e., *$A = \{s : \forall s \in e : \forall e \in E|_\pi\}$. The set of correct traversal initial states I is the subset of A such that $s[pc] = 1$ for every state $s \in I$.*

3.2 Fault model

So far our process model does not contain faults. We now introduce the arbitrary state corruption (ASC) fault model, which is essentially the arbitrary-fault model (Section 2.3) recasted for single-process traversals. In the ASC fault model, faults are state transitions [10] that form a disjunction with the process state transitions [21]. A traversal starts from some initial state and performs a sequence of transitions. Each transition of the traversal is either a *Next* or a *Fault*. Formally, $\Box(\mathit{Next} \lor \mathit{Fault} \lor s' = s)$ holds for every pair of consecutive states $\langle s, s' \rangle$ in any traversal of π.

Figure 2 shows an example of an execution $E_1|_\pi$ forking into execution $E_2|_\pi$ at state s_2 with a *Fault* step. The first traversal of execution $E_1|_\pi$ is the sequence of states $s_1, s_2, \ldots, s_{i+1}$, whereas the first traversal of $E_2|_\pi$ is the sequence of states $s_1, s_2, s_3^2, \ldots, s_{k+1}^2$.[1] In the ASC fault model, a *Fault* step can be a crash transition or an arbitrary process fault, *i.e.*, the corruption of one or more variables in V.

3.2.1 Crash faults

We now model the process faults in the crash-stop fault model, *i.e.*, the crash of a process π running a program p. We assume the existence of a program counter value "halt" different from $0, \ldots, N+1$. The process π performs no further state transitions if pc = "halt".

Definition 18 (Crash fault) *A crash fault is a crash transition that is only enabled if the process is not halted yet. A crash transition sets the program counter to the "halt" value. Formally,*

$$\begin{aligned} \mathit{Crash} &\triangleq pc' = \text{"halt"} \\ \mathit{Halted} &\triangleq pc = \text{"halt"} \\ \mathit{Fault} &\triangleq \neg \mathit{Halted} \land \mathit{Crash} \end{aligned}$$

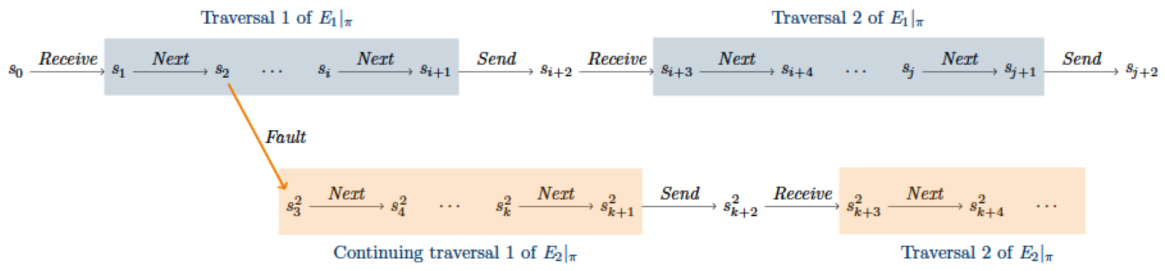
When pc = "halt", the process state transition *Next* is never enabled – *i.e.*, it is always false – because every clause in the disjunction asserts that pc is some value in the set $\{0, \ldots, N+1\}$ (see Definition 15). Since *Next* and *Fault* are disabled if *Halted* is true, the only possible state transitions of process π after a crash are stuttering steps $s' = s$.

Note that Definition 18 is concerned with *process* faults. *Network* faults in the crash-stop fault model – *i.e.*, omission, reordering, duplication, and misrouting – are all captured by the initial state I with faults, as defined in Section 3.2.4. Also note that our notation does not comply with the full TLA+ language [20] when defining state transitions such as *Next* or *Fault*. To simplify the presentation, we often only mention the variables that are modified in the state transition. For example, in *Fault* of Definition 18, pc' is set to "halt", while all other variables in V are kept unmodified. With this simplification, we avoid adding a disjunction to *Fault* with $\forall v \in V \setminus \{pc\} : s'[v] = s[v]$ or some equivalent construction.

3.2.2 Arbitrary process faults

We now define the corruption faults of a process π running a program p. Corruption faults can potentially modify the whole state of process π, and any arbitrary (Byzantine) failure of process π can be modeled as a corruption fault.

[1] We use a superscript to indicate to which execution the state belongs to, *e.g.*, s_3^2 belongs to execution $E_2|_\pi$.



Traversal 1 of $E_1|_\pi$
Traversal 2 of $E_1|_\pi$
$s_0 \xrightarrow{Receive} s_1 \xrightarrow{Next} s_2 \cdots s_i \xrightarrow{Next} s_{i+1} \xrightarrow{Send} s_{i+2} \xrightarrow{Receive} s_{i+3} \xrightarrow{Next} s_{i+4} \cdots s_j \xrightarrow{Next} s_{j+1} \xrightarrow{Send} s_{j+2}$
Fault
$s_3^2 \xrightarrow{Next} s_4^2 \cdots s_k^2 \xrightarrow{Next} s_{k+1}^2 \xrightarrow{Send} s_{k+2}^2 \xrightarrow{Receive} s_{k+3}^2 \xrightarrow{Next} s_{k+4}^2 \cdots$
Continuing traversal 1 of $E_2|_\pi$
Traversal 2 of $E_2|_\pi$

Figure 2: Example of a fault affecting a traversal of execution $E_1|_\pi$. The fault forks $E_1|_\pi$ into another execution $E_2|_\pi$.

Definition 19 (Variable corruption) *A variable corruption changes the value of a variable $v \in V$ to an arbitrary value $x \in D$ such that the new value x is different from the current value $s[v]$. Formally, the formula Corruption defines a variable corruption:*

$$Corruption(v) \triangleq \exists\, x \in D : s[v] \neq x \wedge s'[v] = x$$

In this work, we define a single type of corruption fault: *arbitrary state corruption* (ASC). An ASC fault corrupts a set of variables $W \subseteq V$ (following Definition 19), while leaving all other variables with their previous values.

Definition 20 (Arbitrary state corruption) *An Arbitrary State Corruption (ASC) fault is the corruption of any subset $W \subseteq V$ of variables. Formally,*

$$ASCFault \triangleq \exists\, W \subseteq V : \forall\, w \in W : Corruption(w)$$

One could imagine other types of corruption faults such as the corruption of a single variable, or the corruption of a single bit in a variable. ASC faults result in weaker (*i.e.*, more general) properties, which include more restrictive forms of corruption faults (*e.g.*, Single Event Upset).

Note that we are not interested in the causes of the corruption faults, just on their effect, *i.e.*, the errors caused by them. In our process model, all conditions required by a process to perform a *Next* step are present in its state, so it is natural that faults only corrupt the process state. For example, a fault does not have to compute an addition incorrectly because that is equivalent to performing the addition and corrupting the resulting value in the state afterwards. In this way, ASC faults capture hardware errors in a unified way regardless of whether a hardware error affects memory elements or affects combinational logic circuits that eventually write into memory elements. Also note that any variable $v \in V$ can be corrupt, not only program variables V_p. Since hardening algorithms may only have bookkeeping variables in V, these variables are also subject to corruption.

We now define an arbitrary process fault.

Definition 21 (Arbitrary process fault) *An arbitrary process fault is either a crash transition or an ASC fault. An arbitrary process fault is only enabled if the process is not halted. Formally,*

$$Fault \triangleq \neg Halted \wedge (Crash \vee ASCFault)$$

Transient, intermittent, and permanent faults. In the ASC fault model, corruption faults are transient: they corrupt state variables once they occur, but later writes into the corrupt variables can correct their values. Intermittent and permanent faults are, nevertheless, possible in our fault model. The ASC fault model *per se* does not restrict the frequency in which ASC faults may occur. A permanent or intermittent fault can be modeled as several occurrences of the same type of ASC fault. However, Section 4.1 introduces an upper bound on the frequency of ASC faults to makes hardening more practical.

Faults in the program instructions. In our model, the state of π might be corrupt, but not the program p since p is *not* modeled as part of the process state. A transient corruption of the text segment of a program, *i.e.*, its instructions, typically result in a crash or in an ASC fault, for example, if the corrupted instructions incorrectly modify variables. Evidence from both our experiments and related work [9] suggests that text corruption is mostly harmless, since it quickly leads to the crash of the faulty process.

3.2.3 Corrupt variables and reference state

To reason about the correctness of SEI, we have to precisely define what corrupt and correct variables are. We introduce a specification-only reference state r (following Definition 8). The state r is not accessible by the system and is only used to specify faults and reason about correctness of the algorithms. A variable is said to be corrupt while its value in s is different from its value in r.

Definition 22 (Corrupt and correct variable) *A variable $v \in V$ is corrupt iff the value of v in s is different from the value of v in r; otherwise v is correct. Formally,*

$$\begin{aligned} &\mathit{Corrupt}(v) \triangleq s[v] \neq r[v] \\ &\mathit{Correct}(v) \triangleq \neg \mathit{Corrupt}(v) \end{aligned}$$

Intuitively, the reference state r of process π takes the same process state transitions and the same stuttering steps as state s. Nevertheless, if an arbitrary fault affects the state s, it does not affect the reference state r. Consequently, a corruption fault might make the value of variables in s and r diverge; the error might also be propagated to other variables via *Next* steps. Moreover, a corruption fault might also "fix" a corrupt variable, *i.e.*, the fault might make the value of some variable in s match again the value in r. We now define reference state transitions, reference fault transitions and reference initial states.

Definition 23 (Reference state transition) *The reference state transition RNext is given by Definition 15 using state r instead of s.*

Definition 24 (Reference fault transition) *The reference fault state transition RFault is defined as follows:*

$$\begin{aligned} &\mathit{RFault} \triangleq \\ &\quad \lor s'[pc] = \text{“halt”} \implies r'[pc] = \text{“halt”} \\ &\quad \lor s'[pc] \neq \text{“halt”} \implies r' = r \end{aligned}$$

RFault is a disjunction with two cases. In the first case, if the process halts/crashes, then the next value of *pc* in state r is also set to "halt". In the second case, if a fault different from a crash occurs in s, *RFault* stutters state r.

Definition 25 (Reference initial states I_r) *The same as Definition 17, but where E is the set of all possible executions of the system under crash-stop failures.*

The crash-stop failures in Definition 25 do not only refer to process crashes, but also to network faults considered in the crash-stop model (see Section 2.3). Although the reference initial states have *no* corrupt variables, they might contain misrouted or reordered messages in V_i due to these network faults – remember that we assume the system can tolerate such failures.

3.2.4 Traversals with corruption faults

In general, we have to model and cope with two failure cases illustrated by the example in Figure 3.

Case 1 (direct corruption): A fault directly corrupts variables causing an arbitrary failure. For example, before the *Send* step is taken after state s_{i+1}, a fault corrupts the variables in V_o and the corrupt data is sent out at state s^3_{i+2}.

Case 2 (error propagation): A fault causes a failure indirectly. For example, a fault corrupts variables of the program at state s_2 and the error propagates to variables in V_o via *Next* steps. The process then sends corrupt data out at state s^2_{k+1}.

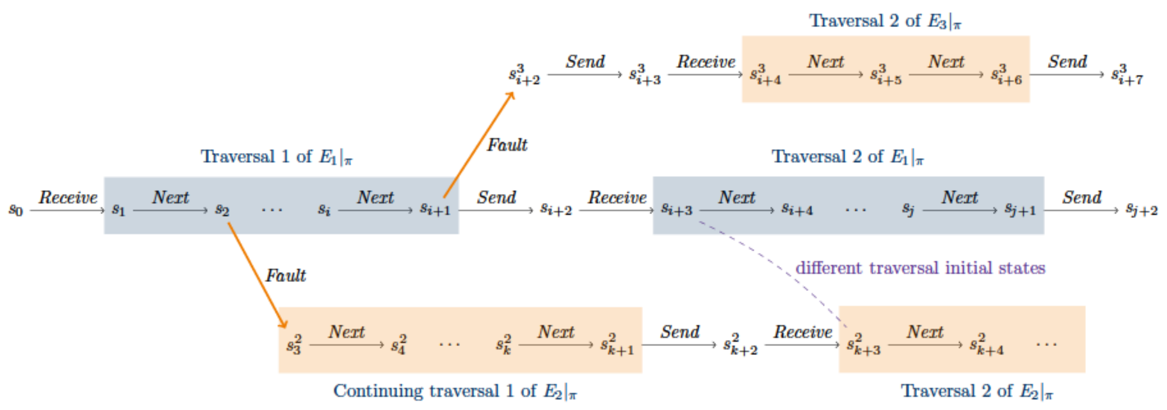

Traversal 2 of $E_3|_\pi$
s^3_{i+2} Send s^3_{i+3} Receive s^3_{i+4} Next s^3_{i+5} Next s^3_{i+6} Send s^3_{i+7}
Fault
Traversal 1 of $E_1|_\pi$
Traversal 2 of $E_1|_\pi$
s_0 Receive s_1 Next s_2 … s_i Next s_{i+1} Send s_{i+2} Receive s_{i+3} Next s_{i+4} … s_j Next s_{j+1} Send s_{j+2}
Fault
different traversal initial states
s^2_3 Next s^2_4 … s^2_k Next s^2_{k+1} Send s^2_{k+2} Receive s^2_{k+3} Next s^2_{k+4} …
Continuing traversal 1 of $E_2|_\pi$
Traversal 2 of $E_2|_\pi$

Figure 3: Example of a fault affecting execution $E_1|_\pi$ inside and outside a traversal. The faults fork $E_1|_\pi$ into other execution.

Note that Case 1 can also occur during a traversal, instead of occurring after a traversal. Imagine a variable $v \in V_o$ is only modified by a *Next* step from state s_1 to s_2. In a fault-free execution, v is correct when sent out in the *Send* step at state s_{i+1}. However, if a *Fault* step corrupts v at state s_2 and no *Next* step modifies v until s^2_{k+1}, then v is corrupt when sent out in the *Send* step from s^2_{k+1} to s^2_{k+2}.

Also note that Case 2 might become a failure only in some further traversal, for example, at the end of traversal 2 of execution $E_2|_\pi$. Consequently, the traversal initial states of traversal 2 in execution $E_1|_\pi$ and in execution $E_2|_\pi$ are different. Therefore, to model all possible traversals, we not only add *Fault* steps to the set of state transitions of the process, but we also add faults in the traversal initial states, so that at least one of the initial states contains the variable corruption of traversal 1 in $E_1|_\pi$.

We now reformulate the definition of a traversal (Definition 16) to take faults and the reference state into account. We join the process state transitions with the reference state transitions, *i.e.*, $(Next \land RNext)$. Since faults do not corrupt the reference by definition, we join the fault transitions with reference fault transitions, *i.e.*, $(Fault \land RFault)$. State transitions always satisfy the following formula:

$$\Box\big((Next \land RNext) \lor (Fault \land RFault) \lor (s' = s \land r' = r)\big)$$

Since the previous definition of traversal initial state (Definition 17) does not consider corruption faults, we redefine the traversal initial state as follows.

Definition 26 (Traversal initial states I_s with faults) *A traversal initial state s is any state that is either correct,* i.e., *equal to the reference initial state $r \in I_r$ of the traversal, or some variable v is such that $s[v] \neq r[v]$. The set of initial values I_s boils down to the set $D^{|V|}$ of all possible states for a set of variables V – remember that D is the domain of values of all variables.*

With reference state transitions *RNext*, the set I_r of reference initial states, and the set I_s of traversal initial states, we can redefine traversals by coupling the transitions on r and on s.

Definition 27 (Traversal with fault and reference state) *A traversal with faults and reference state of process π running a program p is an execution starting from a potentially corrupt initial state followed by states satisfying the process state transitions Next and the reference state transitions RNext, or the fault state transitions Fault and the reference fault transitions RFault, or a stuttering step. Formally,*

$$\begin{aligned}
&Traversal \triangleq \\
&\quad \land\, r \in I_r \land s \in I_s \\
&\quad \land\, \Box\big((Next \land RNext) \lor (Fault \land RFault) \lor (s' = s \land r' = r)\big)
\end{aligned}$$

In summary, the initial reference state r is one of the possible reference initial states of Definition 25. The initial state s is one of the possible states in I_s – being either equal to r or corrupt. If a process state transition (*Next* step) takes place, then a reference state transition (*RNext* step) takes place as well. If a fault transition (*Fault* step) takes place, then a reference fault transition (*RFault*) takes place. Finally, if the state s stutters, the reference state r stutters.

Remark on fault assumptions. Note that without any further restrictions, corruption faults can change the variables in any way and arbitrarily often, including the initial state of the traversal. Definition 27 allows for ASC faults to induce a process π to commit Byzantine failures, *i.e.*, sending corrupt but valid messages. To work correctly, SEI requires a small set of fault assumptions, which we introduce in Section 4.1.

Remark on stuttering traversals. If an ASC fault causes the program counter to point $N+1$ before any variable is modified, this traversal is said to *stutter due to a fault.* A traversal that stutters due to a fault simply results in message omissions – *i.e.*, an input message not being processed – and/or message duplication – *i.e.*, an output message being sent again. Such failures can be mapped as network faults in the distributed system model. Hence, we do not consider traversals that stutter due to faults in our process fault model.

3.2.5 Relation between corrupt variables and corrupt messages

We conclude this section by showing that if a process π sends a corrupt message (*cf.* Definition 5) then there is at least one variable $v \in V_o$ that is corrupt (*cf.* Definition 22). Lemma 1 allows us to reason exclusively on corrupt variables in the design of SEI in the next sections.

Lemma 1 (Corrupt message implies corrupt V_o) *Let E be an execution of the distributed system and $\sigma = \langle T_1, T_2, \ldots \rangle$ the sequence of traversals executed by process π in E that modified a variable in the state,* i.e.*, we ignore stuttering traversals. Let m_o^k be an output message sent by process π in a traversal $T_k \in \sigma$. If message m_o^k is corrupt, then there exists $v \in V_o$ such that $Corrupt(v)$ holds at the end of T_k.*

Proof:
1. Let s_e be the state at the end of traversal T_k such that $s_e[pc] = N$.
2. Let r_e be the reference state at the end of traversal T_k such that $r_e[pc] = N$.
3. By transposition, it is sufficient to show that if $\forall v \in V_o : \neg Corrupt(v)$ at state s_e, then m_o^k is a correct message according to Definition 4, *i.e.*, m_o^k has a correct generation history.
4. Assume $\forall v \in V_o : s_e[v] = r_e[v]$ according Step 3 and definition of $Corrupt(v)$ (Definition 22).
5. Let r_i^j be the reference initial state of a traversal T_j.
6. Let m_i^j be the input message represented by the values of V_i in state r_i^j.
7. $h_k = \langle m_i^1, m_i^2, \ldots, m_i^k \rangle$ is a correct generation history h_k for message m_o^k.
   - 7.1. h_k is a generation history for message m_o^k, by Definitions 2 and 27 since we only consider non-stuttering traversals.
   - 7.2. The precedence relationship of Definition 1 is a total order, since π is single-threaded and deterministic.
   - 7.3. CASE: $k = 1$
     - 7.3.1. π sends no output message before m_o^1, therefore h_k is correct.
   - 7.4. CASE: $k > 1$
     - 7.4.1. Let message m_o^j be the correct message sent by π in a traversal T_j such that there is no m_o that precedes m_o^k but does not precede m_o^j.
     - 7.4.2. By induction, $h_j = \langle m_i^1, m_i^2, \ldots, m_i^j \rangle$ is a correct generation history for m_o^j.
     - 7.4.3. $h_k = \langle m_i^1, m_i^2, \ldots, m_i^j, m_i^k \rangle$ extends h_j, therefore it is correct. □

4 Single-threaded SEI-hardening

In this section, we present the SEI-hardening of single-threaded programs. The extension to multiple threads is presented in Section 5. The single-threaded SEI-hardening achieves error isolation and can leverage encoded replica state to save space. We first refine the process and fault models we have defined in Section 3, follow it with a presentation of SEI-hardening, and prove its correctness.

4.1 Model refinements

SEI-hardening differentiates between global and local variables. Intuitively, global variables are variables in the main memory that might be allocated throughout multiple traversals, whereas local variables are registers holding values temporarily. We differentiate between local and global variables in this context to represent the flow of values when performing a computation. When incrementing the value of a global variable v, for example, the value of v is copied to a register (some local variable v_l), which is then incremented.

A more precise definition of global and local variables is as follows:

Definition 28 (Global variables) *The set $V_g \subset V$ is the set of global variables of process π. Global variables are long-lived variables, whose values are used in multiple traversals of process π. The global variables are initialized with default values in the initial state of the first traversal of π, and their values depend on the sequence of traversals.*

Definition 29 (Local variables) *The set $V_l \subset V$ is the set of local variables of π, where $V_l \cap V_g = \{\ \}$. Local variables are short-lived variables,* e.g., *registers, whose values are discarded once a traversal has finished. Local variables are initialized with default values before being used in each traversal.*

Following a RISC-like architecture, we assume all computations are performed on local variables, and global variables are only accessed via load and store operations.

Definition 30 (Load and store operations) *Global variables can only be accessed via Ld and St operations:*

- *$Ld(v_d, v_s)$ loads the value of global variable v pointed by a local variable v_s,* i.e., *$v = s[v_s]$, into a local variable v_d.*
- *$St(v_d, v_s)$ stores the value of a local variable v_s into a global variable v pointed by a local variable v_d,* i.e., *$v = s[v_d]$.*

Moreover, only global variables can be accessed via Ld and St operations, i.e., *a local variable $v_1 \in V_l$ cannot point to another local variable $v_2 \in V_l$.*

The restriction of *Ld* and *St* operations being only used to access global variables helps us to precisely define the transformations rules in the next section. In practice, however, memory locations can also be used as local variables, *e.g.*, the program stack. In fact, our compiler-based implementation of SEI considers variables in the stack – which are stored in the main memory – as a set of local variables, and differentiates at run time whether the *Ld* or *St* operation is accessing a global variable in the main memory, or a local variable in the main memory.

In our algorithms, we make use of the *Abort* operation defined as follows.

Definition 31 (*Abort* operation) *The Abort operation sets the pc with "halt" (see Page 10), forcing the process to stop performing further Next steps.*

4.1.1 Corruption coverage

The goal of SEI is to guarantee Properties 2, 3, and 4 in face of corruption faults. In particular, guaranteeing Property 3 is the major challenge since it asserts what a *faulty* process is allowed to do. Local error exposure (Property 3) asserts that if a faulty process sends a corrupt message out, this message is invalid. Consider again the example in Figure 3. SEI should guarantee that this property holds whenever a traversal terminates, *e.g.*, at states s_{i+1}, s^2_{k+1}, s^3_{i+6}, etc. Without any further assumption, however, corruption faults can change the variables in any way and arbitrarily often, inducing a process π to commit Byzantine failures, *i.e.*, sending corrupt but valid messages. Therefore, we have to restrict *Fault* steps and faults in the initial state with *fault assumptions*, otherwise the traversal might lead to or even already start at a state in which local error exposure can be violated.

The first fault assumptions SEI relies upon is *corruption coverage*[2]. Corruption coverage inhibits the possibility of "direct corruptions" (Case 1 in Page 12) becoming arbitrary failures. The intuition behind corruption coverage is that a hardening technique can employ some form of space redundancy to protect a subset of variables of V. SEI protects all variables in $V_g \subset V$, *i.e.*, the set of global variables, and consider the sets V_i and V_o to be part of V_g. For each (protected) variable in V_g, SEI reserves a *replica variable*, as we explain in Section 4.1.2. Corruption coverage then asserts that every variable in V_g modified by a *Fault* step at state s is invalid at state s'.

[2]Corruption coverage is called *fault diversity* in the terminology of Correia *et al.* [9].

Assumption 2 (Corruption coverage) *A Fault step at some state s satisfies corruption coverage if and only if it is a crash transition or if every variable $v \in V_g$ modified during the fault is invalid at state s'. Formally, the* ASCFault *(Definition 20) formula is composed via conjunction with the following formula:*

$$\mathit{CorruptionCoverage} \triangleq \forall v \in V_g : s'[v] \neq s[v] \Longrightarrow \neg \mathit{Valid}'(v)$$

where the primed formula $\mathit{Valid}'(v)$ is the formula $\mathit{Valid}(v)$ with state s'.

The concept of message validity (Definition 6) states that messages are sent out with enough information to be classified as valid or invalid by a correct receiver. Since messages are stored in the state as variables in V_i and V_o, and V_i and V_o are part of V_g, it is natural to use the space redundancy in the process state – *i.e.*, the replica variables of V_g – to also protect messages in an end-to-end fashion. Consequently, corruption coverage guarantees that, if a message is corrupted in the network, then some variables representing the message are invalid upon receipt. Also, if a fault directly corrupts some variable $v \in V_o$ at some state s, then v is invalid at state s', implying that the message $m \notin CV$. In our example, a fault directly corrupting a variable $v \in V_o$ at state s_{i+1} results in an invalid variable v at state s^3_{i+2}. If the message represented by V_o is sent out at state s^3_{i+2}, then a correct process receiving the message can discard it. Corruption coverage allows the hardening techniques to focus on error propagation (see Page 12) since direct corruptions cannot result in corrupt messages by assumption.

Corruption coverage also has to hold at the traversal initial states, rendering a corrupt message in V_i invalid. We redefine the set of traversal initial states as a function of the reference initial state r as follows.

Definition 32 (Traversal initial states $I_s(r)$ with faults) *A traversal initial state s is any state that is either correct,* i.e., *equal to the reference initial state $r \in I_r$ of the traversal, or for every variable v such that $s[v] \neq r[v]$, v is invalid in s. Let $D^{|V|}$ represent all possible states for a set of variables V. The set of all possible traversal initial states $I_s(r) \subseteq D^{|V|}$ is formally defined as follows.*

$$I_s(r) \triangleq I_r \cup \{s \in D^{|V|} : \forall v \in V_g : s[v] = r[v] \vee \neg \mathit{Valid}(v)\}$$

4.1.2 Replica variables

SEI employs space redundancy to detect faults. We now define the set of replica variables V_r and related sets of variables V_s and V_h. Next, we define $\mathit{Valid}(v)$ for $v \in V_g$.

Definition 33 (Replica variables V_r) *The set of variables $V_r \subset V$ is such that $V_r \cap V_g = \{\ \}$, $V_r \cap V_l = \{\ \}$, and $|V_r| \geq |V_g|$.*

Definition 34 (Replica mapping μ_r) *There is an injective function $\mu_r : V_g \to V_r$ that maps each variable in V_g to a distinct variable in V_r. If a variable $v \in V_g$ and a variable $\bar{v} \in V_r$ are such that $\mu_r[v] = \bar{v}$, then v and $\bar{v}$ are called replica variables.*

Figure 4 depicts the main sets of variables of a process π. By definition, global variables and only global variables have replicas. The set of replica variables V_r is neither global nor local. In practice, replica variables are most likely to be hosted in the main memory just as global variables. Nonetheless, the replica of a variable does not need to be a complete copy: ECC in main memory and CRC for messages are examples of redundant information, in hardware or software, that can be used to implement replicas in reduced space.

Recall that the set of all variables of program p running on process π is $V_p \subset V$ (see Section 3.1). We define parts of V_g and, indirectly, of V_r to be exclusive for the program p or for bookkeeping of SEI.

Definition 35 (Program state) *The set $V_s = V_p \cap V_g$ is the set of all global variables used in the program p running on process π. The part of the state formed by the variables V_s is referred to as the program (global) state.*

Definition 36 (Hardening state) *The set $V_h \subset V_g$ is the set of all global variables used by SEI-hardening. The set V_h is such that $V_h \cap V_s = \{\ \}$. The part of the state given by variables V_h is referred to as the hardening state.*

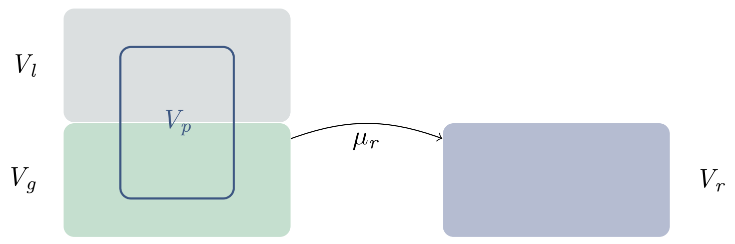


Figure 4: Variables of process π. The set of all variables is $V = V_l \cup V_g \cup V_r$. The set of all variables of program p running on process π is V_p. The set of all global variables of program p running on process π is $V_s = V_p \cap V_g$.

We model the variables representing input and output messages (*i.e.*, V_i and V_o, respectively) as subsets of V_s since, in practice, messages are part of the program variables and are typically stored in main memory. As discussed in Section 2.4.2, we assume that an error-detection code is transmitted along with each message sent over the network. Since the sets V_i and V_o, being part of the global variables, have counterpart replica variables in V_r, one could be inclined to directly use those replica variables as error-detection code for the messages. In practice, however, if the replica variables are implemented with ECC memory, then a SEI-hardening implementation cannot retrieve the values of these replica variables when sending a message since the hardware does not allow direct access to the ECC memory. Moreover, a SEI-hardening implementation cannot directly write the replica variables (*i.e.*, the error-detection code or CRC) received from network into the ECC memory. Therefore, we model a second set of replica variables exclusive for variables in $V_i \cup V_o$. This second set of variables can be a part of V_g or V_l depending on the implementation.

Definition 37 (Replicas of V_i and V_o) *The set of variables $V_i \subset V$ represents the input message, with $V_i \subset V_s$. The set of variables $V_o \subset V$ represents the output messages, with $V_o \subset V_s$. Each variable v in $V_i \cup V_o$ has two replica variables: $\bar{v} \in V_r$ and $\ddot{v} \in V$. The set $\bar{V}_i$ and $\bar{V}_o$ represent the first set of replica variables of V_i and V_o, respectively. The set $\ddot{V}_i$ and $\ddot{V}_o$ represent the second set of replica variables of V_i and V_o, respectively.*

Notation of replica variables. We use a bar on top of a variable v to represent its replica in V_r: the bar can be thought as the complement of v. We use a double dot on top of a variable v to represent v's second replica (if v is an input or output message): the double dot can remind that $\ddot{v}$ is the *second* replica of v. In Table 2, we list all variables sets used in our model.

One of the fault assumptions of SEI-hardening is corruption coverage (Assumption 2). Corruption coverage asserts that if a global variable is corrupted by a fault, its replica does not have the same value at some state s, *i.e.*, $\neg Valid(v)$ does not hold at s. Defining $Valid(v)$ for variables in $V_g \backslash (V_i \cup V_o)$ is straightforward: $Valid(v) \triangleq s[v] = s[\bar{v}]$. If a fault corrupts a variable $v \in V_g \backslash (V_i \cup V_o)$ at state s, then at state s' it holds that $s'[v] \neq s'[\bar{v}]$ by corruption coverage. Remember that s' is the state immediately following s in the process execution.

In contrast, defining $Valid(v)$ for input and output variables requires more care since such variables have two replicas. Corruption coverage guarantees that $Valid(v)$ does not hold after a fault. One may try to define $Valid(v) \triangleq s[v] = s[\bar{v}] \wedge s[v] = s[\ddot{v}]$. Such a definition does not reflect the intuitive notion of corruption coverage because after a fault corrupting some variable $v \in V_o$, it could hold that $s[v] \neq s[\bar{v}]$ and $s[v] = s[\ddot{v}]$. Unfortunately, it violates local error exposure (Property 3) since a *Send* step would send v and $\ddot{v}$ out. Even if the SEI-hardening checks whether $s[v] = s[\bar{v}]$ before sending out a message containing v, this check could be skipped by a control-flow fault (SEI-hardening allows faults to affect the control-flow, as we will see below). Therefore, we define the validity of V_i and V_o as $Valid(v) \triangleq s[v] = s[\bar{v}] \vee s[v] = s[\ddot{v}]$ because the negation of $Valid(v)$ gives the corruption coverage property we intuitively search for.

Definition 38 (*Valid*(v) **predicate)** *Valid*(v) *is defined for each subset of* V_g *as follows.*

$$\begin{aligned}
\mathit{Valid}(v) &\triangleq s[v] = s[\bar{v}] && \text{for all } v \in V_g \setminus V_i \cup V_o \\
\mathit{Valid}(v) &\triangleq s[v] = s[\bar{v}] \vee s[v] = s[\ddot{v}] && \text{for all } v \in V_i \cup V_o \\
\mathit{Valid}(\bar{v}) &\triangleq \mathit{Valid}(v) && \text{for all } \bar{v} \in V_r \\
\mathit{Valid}(\ddot{v}) &\triangleq \mathit{Valid}(v) && \text{for all } \ddot{v} \in \ddot{V}_i \cup \ddot{V}_o
\end{aligned}$$

For every global variable v that does not belong to V_i or V_o, v is valid at a state s if and only if *Valid*(v) holds, *i.e.*, v equals $\bar{v}$. In case v is an input or output variable, *Valid*(v) also holds if v equals the second replica $\ddot{v}$. If a variable $v \in V_i \cup V_o$ is equal to only one replica, *e.g.*, $\bar{v}$, we say that v is valid with respect to $\bar{v}$ and invalid with respect to $\ddot{v}$. We also define *Valid*(v) for replica variables ($\bar{v}$ and $\ddot{v}$), although we do not use the predicate directly in our proofs. This definition is important because an ASC fault could corrupt only the replica variables of a variable v, without modifying v itself.

An ASC fault can never turn a variable to become valid again, *i.e.*, $\neg\,\mathit{Valid}(v)$ holds from immediately after a fault corrupts a variable v until v is assigned to a new value by some instruction of program p or by the *Receive step*.

4.1.3 Fault frequency

Besides assuming corruption coverage, SEI-hardening assumes an upper bound on the frequency of fault occurrences within a traversal. An execution of the system – *i.e.*, a sequence of traversals – can observe an unbounded number of faults. In particular, we do not assume any bound on the number of faults between traversals.

Assumption 3 (Fault frequency) *A single fault might occur in a traversal A of a process π.*

Fault frequency is a system-specific assumption because the execution time of handler programs and even the execution time of single instructions changes from system to system. Nevertheless, we target distributed systems which handle events in less than milliseconds, which is a reasonable fault frequency bound considering the error rates published in the literature [9]. The fault model consequently assumes that no two faults occur within such a short time window. The frequency of *uncorrectable* hardware-level data corruption reported by studies "in the wild" indicate that this assumption holds with very high probability [26, 29, 18].

A consequence of assuming fault frequency is that SEI cannot tolerate permanent hardware faults (see discussion on Page 11 of Section 3.2). The fault frequency assumption also renders SEI vulnerable to faults affecting the text segment, *i.e.*, the program instructions, because such faults behave as permanent faults. Evidence suggests, however, that faults corrupting the text segment quickly lead to a process crash [16]. Moreover, such faults are innocuous if hardware error detection is employed (over the whole memory hierarchy) because every loaded instruction is compared to its replica when the processor fetches the instruction.

4.1.4 Pointer corruptions

In SEI-hardening, we assume that the use of corrupt pointers invariably leads to a process crash. If a load or store operation is executed using a corrupt local variable as a pointer, *i.e.*, the corrupt local variable points to some arbitrary value in the memory picked at random by a fault, then the process crashes.

Assumption 4 (Benign pointer corruption) *If a corrupt local variable $v_l \in V_l$ is used in a Ld or St operation, then π immediately crashes.*

Assumption 4 asserts that pointer corruption faults result in crashes with such a high probability that we can consider negligible the probability of π not crashing when using a corrupt pointer in a load or store operation. This assumption is extremely important in SEI-hardening. If pointer corruptions were allowed, we would have to reason about the effects of error propagation to any variable in the program. Assumption 4 restricts the error propagation to those variables intended to be written or read in the algorithms.

Since pointers cannot be corrupt, we can define high-level read and write operations to simplify the presentation of our algorithms. These operations refer directly to a global variable instead of referring to a local variable that points to the global variable. In an implementation of SEI, however, there is such a local variable pointing to the global variable.

Definition 39 **($v \rightarrow v_l$)** *The read operation $v \rightarrow v_l$ loads the value of the global variable $v \in V_g$ into the local variable $v_l \in V_l$.*

Definition 40 **($v \leftarrow v_l$)** *The write operation $v \leftarrow v_l$ stores the value of the local variable $v_l \in V_l$ into the global variable $v \in V_g$.*

Note that Correia *et al.* achieve a similar simplification in PASC with their *access fault* assumption [8].

4.1.5 Further assumptions

Finally, we introduce the last two assumptions of SEI-hardening, which are also used in PASC.

Assumption 5 (Immutable input messages) *Input messages are only read and never modified. In other words, no variable $v \in V_i$ is ever modified within a traversal.*

Assumption 6 (Initially correct state) *The first state of the first traversal of an execution of a process π is correct,* i.e., *$s = r$.*

4.2 SEI-hardening specification

We now present the specification of SEI-hardening. SEI-hardening transforms the program p running on a process π of a crash-stop tolerant distributed system into a hardened program p_h. By hardening the programs running on all processes of the system, the system can tolerate arbitrary faults in non-malicious environments (see Section 2.4.2). The SEI-hardening transformation mainly duplicates the execution of p and inserts several checks to guarantee error isolation.

SEI does not define how the transformation is to be implemented. In particular, our specification does not dictate whether the implementation realizes replicas using software-based error-detection codes or leveraging existing hardware error detection mechanisms.

4.2.1 Blocks and gates

In SEI, a program is divided in blocks and gates. *Blocks* are sequences of instructions implementing either the original functionality in program p, or implementing part of the hardening. *Gates* are also sequences of instructions, but they serve to check the control flow of the traversal only.

The first transformation rule of SEI defines the structure of a hardened program, independently of the program p itself.

Rule 1 (Hardened program structure) *A hardened program p_h is a sequence of gates and blocks as shown in Algorithm 1.*

A traversal of a SEI-hardened program p_h is an interleaved execution of blocks and gates starting with the block *Filter*. Line 84 represents the end of the handler program, *i.e.*, once the execution reaches Line 84, messages might be sent out (see process model in Section 3.1). The gates guarantee that the high-level control flow executes correctly, *i.e.*, that the blocks *Filter*, *Prepare*$_1$, *Prepare*$_2$, *Exec*$_1$, *Reset*, *Exec*$_2$, and *Validate* execute in the order given by Algorithm 1. As we will describe below, *FirstGate*, *Gate*(...) and *LastGate* are the procedures given by Algorithm 8. We first focus on the presentation of the blocks assuming blocks execute in order, and ignoring the existence of the gates. We describe the gate algorithms later in this section.

Algorithm 2 describes the three *Initialization* blocks. The *Filter* block checks whether the input message is valid, discarding it in case it is invalid by jumping to the first instruction after the program p_h (Line 84). The *Prepare*$_1$ block initializes the bookkeeping data structures used by SEI. The *Prepare*$_2$ performs reinitializes all data structures. The repetition of the operations guarantees that if the *Prepare*$_1$ block is skipped by a fault, the data structures are still correctly initialized. Note that if *Filter* is skipped by a fault, a validity check in *Validate* aborts process π.

Once the input message is checked, and the data structures are initialized, the *Exec*$_1$ block executes program p for the first time (Algorithm 1). Subsequently, the *Exec*$_2$ block executes program p for the

```
    program $p_h$
1       $Filter$
3       $FirstGate$
9       $Prepare_1$
13      $Gate(cf_p, cf_g)$
18      $Prepare_2$
22      $Gate(cf_1, cf_p)$
*27     $Exec_1$
28      $Gate(cf_r, cf_1)$
33      $Reset$
48      $Gate(cf_2, cf_r)$
*53     $Exec_2$
54      $Gate(cf_c, cf_2)$
59      $Validate$
73      $LastGate$

84
```

Algorithm 1: Hardened program p_h

```
    block $Filter$
1       if $\neg CheckMessage(V_i)$ then
2           goto 84

    block $Prepare_1$
9       foreach $v \in V_s$ do
10          $\mathring{O}(v) \leftarrow_{v\bar{v}}$ FALSE
11          $\mathring{N}(v) \leftarrow_{v\bar{v}}$ FALSE
12          $U(v) \leftarrow_{v\bar{v}}$ FALSE

    block $Prepare_2$
18      foreach $v \in V_s$ do
19          $\mathring{O}(v) \leftarrow_{v\bar{v}}$ FALSE
20          $\mathring{N}(v) \leftarrow_{v\bar{v}}$ FALSE
21          $U(v) \leftarrow_{v\bar{v}}$ FALSE
```

Algorithm 2: *Initialization* blocks: *Filter*, *Prepare*$_1$, and *Prepare*$_2$

```
    function $CheckMessage(V_c)$ do
+0      foreach $v \in V_c$ do
+1          if $v \neq \bar{v}$ or $v \neq \ddot{v}$ then
+2              return FALSE

+3      return TRUE
```

Algorithm 3: Validity check of input variables ($V_c = V_i$) or output variables ($V_c = V_o$)

```
    function $Check(V_c)$ do
+0      foreach $v \in V_c$ do
+1          if $v \neq \bar{v}$ then
+2              return FALSE

+3      return TRUE
```

Algorithm 4: Validity check of a set of variables $V_c \subseteq V_g$

second time. Blocks *Exec*$_1$ and *Exec*$_2$ are transformed versions of the original program p using Rules 2-6. Between *Exec*$_1$ and *Exec*$_2$ blocks, the *Reset* block rolls back all changes done in *Exec*$_1$ to variables in V_s, so that *Exec*$_2$ can repeat the same computation. Finally, the *Validate* block compares the computations of *Exec*$_1$ and *Exec*$_2$ performing a series of checks.

We do not restrict how blocks and gates are implemented – as, for example, procedures, functions, or macros. To ease the presentation, blocks and gates are all inlined to form a single line count over all blocks (the numbers on the left margin of the algorithms). Some algorithms are used multiple times, not having an absolute line number. In such algorithms, we prefix the line numbers with a "+" sign. Blocks *Exec*$_1$ and *Exec*$_2$ count for a single line – marked with a "*" sign in Algorithm 1 – since their length depends on the specific program p being used. Finally, lines of the algorithms may represent several instructions, *e.g.*, if-then structures. "The execution of a line" means that the process takes enough *Next* steps so that all instructions of the line are executed. We *do not* consider all instructions of a line to be executed atomically with respect to *Fault* steps.

4.2.2 Transferring trust from and to message replicas

As discussed in Section 4.1, with each message m sent over the network, a replica of m (possibly in the form of an error-detection code such as CRC) is sent along to detect data corruption. Upon receiving a message m, process π writes m into the variables of V_i and $\bar{V}_i$. Moreover, π writes the replica of message m – *i.e.*, the error-detection code – into the second replica variables $\ddot{V}_i$. Intuitively, the double check in *Filter* (Algorithm 2, Line 1) and *Validate* (Algorithm 5, Line 69) "transfers" the validity from the replica variables $\ddot{V}_i$ to the replica variables $\bar{V}_i$. Lemma 2 formalizes this intuition.

Algorithm 3 represents the validity check for messages, whereas Algorithm 4 represents the validity check for any other variable in V_g. The ability to check the validity of variables quickly is essential in SEI-hardening because these checks are performed very often. To support different implementations, we do not specify how exactly the validity check is implemented. If the system leverages existing hardware error-detection codes, then the check $v \neq \bar{v}$ is automatically performed by the hardware with any ordinary load operation of v. Once the validity of the input message has been "transferred", the variables in V_i

can simply be checked against the variables in $\bar{V}_i$ as normal variables; having the benefit of fast checks if these are used in the implementation.

4.2.3 Data structures

SEI uses three data structures to bookkeep changes to variables in V_s:

- O is a set data structure representing the *old* values of variables in V_s modified during Exec_1.
- N is a set data structure representing the *new* values of variables in V_s modified during Exec_1.
- U is a map data structure marking the variables in V_s modified (*updated*) during Exec_2.

O, N and U are instances of two simple data structure types: maps and sets. We model maps and sets as follows.

Definition 41 (Map data structure) *A map data structure D is a set of global variables $V_D \subset V_g \setminus V_s$ and a bijective function $\mu_D : V_s \to V_D$.*

At each state s, a map data structure implements an in-memory function that *maps* each variable v in V_s to some value (possibly different than $s[v]$). For any $v \in V_s$, we use the shorthand notation "$D(v)$ at state s" meaning the value of $\mu_D[v]$ at state s, *i.e.*, $s[\mu_D[v]]$.

Definition 42 (Set data structure) *A set data structure D is composed of a map data structure D and an auxiliary map data structure $\mathring{D}$. For all $v \in V_s$, the variable $\mu_D[v]$ stores a value for variable v. For all $v \in V_s$, if the value of the variable $\mu_{\mathring{D}}[v]$ is* TRUE, *the set data structure is said to contain v, otherwise v is not in the set data structure.*

When $s[\mu_{\mathring{D}}[v]] = \text{TRUE}$, the set contains a value for variable v, *i.e.*, the value $s[\mu_D[v]]$. When $s[\mu_{\mathring{D}}[v]] = \text{FALSE}$, the set does not contain a value for variable v, and the value $s[\mu_D[v]]$ is undefined. For any $v \in V_s$, we use the shorthand notation:

- "$v \in D$ at state s" meaning that $\mathring{D}(v)$ at state s is TRUE, *i.e.*, $s[\mu_{\mathring{D}}[v]] = \text{TRUE}$; and
- "$v \notin D$ at state s" meaning that $\mathring{D}(v)$ at state s is FALSE, *i.e.*, $s[\mu_{\mathring{D}}[v]] = \text{FALSE}$.

Data structures are initialized in blocks $\mathit{Prepare}_1$ and $\mathit{Prepare}_2$. They are initialized twice because a partial initialization can compromise the correctness of the hardening. Clearly, these definitions of data structures are rather space inefficient. In a real implementation, these data structures would not contain a mapping for all variables in V_p, but instead they would dynamically adapt to the currently used variables. The definitions above however simplify our formalization. They are general enough to represent many real implementations of set and map data structures because they require multiple instructions to introduce elements in the set or map. These instructions are not atomic with respect to *Fault* steps.

4.2.4 High-level assignments

We now define two high-level assignment operations used in the algorithms of blocks and gates. These operations write a value to or copy the value between variables in V_g. The operations writing any variable $v \in V_g$ also write in its replica variable $\bar{v} \in V_r$.

Definition 43 ($v \leftarrow_{v\bar{v}} v_l$) *Given a global variable $v \in V_g$ and a local variable $v_l \in V_l$, the high-level instruction $v \leftarrow_{v\bar{v}} v_l$ writes the value of v_l into the variable v and its replica $\bar{v}$.*

```
+0  $v \leftarrow v_l$
+1  $\bar{v} \leftarrow v_l$
```

Since v_l is a local variable, v_l has no replica. A corruption of v_l can cause v and its replica $\bar{v}$ to be corrupt and valid, *i.e.*, equal. In our algorithms, we also use $v \leftarrow_{v\bar{v}}$ TRUE or $v \leftarrow_{v\bar{v}}$ FALSE, where TRUE and FALSE are constants. In such cases, a fault cannot make both replicas have the same value since constants are part of the instruction, and assume no faults occur in the text segment by definition (see discussion in Section 3, Page 12).

Definition 44 **($v \leftarrow_{Cpy} w$)** *Given two global variables $v, w \in V_g$, the high-level instruction $v \leftarrow_{Cpy} w$ copies the value of w into v and its replica $\bar{v}$. We store the value of w temporarily in the local variable $v_l \in V_l$. The high-level assignment $v \leftarrow_{Cpy} w$ is the following sequence of instructions:*

```
+0  $w \rightarrow v_l$
+1  $v \leftarrow_{v\bar{v}} v_l$
```

Note that, in a real computer equipped with ECC in its memory modules, copying the value of a memory location w to another location v does not automatically copy w's error code into v's error code. Our modeling of copying values between global variables captures this window of vulnerability by first copying the value of w into a local variable v_l. Since v_l is a local variable, a corruption of v_l can then propagate to v and $\bar{v}$. SEI can detect such corruptions, as proved below.

4.2.5 Transformation of program p into blocks $Exec_1$ and $Exec_2$

Block $Exec_1$ is the original program p with Rules 2, 3, and 5 applied, whereas block $Exec_2$ is program p with Rules 2, 4 and 6 applied. These rules substitute read ($v \rightarrow v_l$) and write instructions ($v \leftarrow v_l$) in program p with sequences of instructions that additionally check the validity and manipulate the bookkeeping data structures.

Rule 2 **(Reading variable $v \in V_s$ in blocks $Exec_1$ and $Exec_2$)** *A read instruction $v \rightarrow v_l$, with global variable $v \in V_s$ and local variable $v_l \in V_l$, is substituted by the following sequence of instructions.*

```
     replace $v \rightarrow v_l$ with
+0       if $\neg Check(v)$ then
+1           $Abort$
+2       $v \rightarrow v_l$
```

The validity check of variable $v \in V_s \cup V_i$ is performed only the first time v is read in $Exec_1$ and the first time v is read in $Exec_2$.

Before a variable is read, its validity is verified by comparing it with its replica. This verification is cheap if we use hardware memory protection. If software-level memory protection is used instead, it is sufficient to do the check only the first time a variable is read in the block. In fact, in this case the cost of verifying if the variable has been already read can be lower than the cost of executing the comparison.

Rule 3 **(Writing variable $v \in V_s$ in blocks $Exec_1$)** *A write instruction $v \leftarrow v_l$, with global variable $v \in V_s$ and $v_l \in V_l$, is substituted by the following sequence of instructions.*

```
      replace $v \leftarrow v_l$ with
+0        if $v \notin O$ then
+1            if $\neg Check(v)$ then
+2                $Abort$
+3            $O(v) \leftarrow_{Cpy} v$
+4            if $\neg Check(\mathring{O}(v))$ or $v \in O$ then
+5                $Abort$
+6            $\mathring{O}(v) \leftarrow_{v\bar{v}}$ TRUE
+7            if $\neg Check(v)$ or $v \neq O(v)$ then
+8                $Abort$
+9        $v \leftarrow_{v\bar{v}} v_l$
+10       if $\neg\check{(}\mathring{O}(v))$ or $v \notin O$ then
+11           $Abort$
```

The sequence of instructions from Line +3 to +6 add the value of v to the set O. In particular, the value of v in the data structure O is up-to-date after Line +3, but v is only contained by O, *i.e.*, $v \in O$, from the state after the execution of Line +6. The check at Line +5 guarantees no control-flow error can force a second execution of Line +3. The check at Line +8 guarantees that the validity of v is propagated to $O(v)$.

Write assignments during the second execution are simpler than in the first execution:

Rule 4 (Writing variable $v \in V_s$ in blocks $Exec_2$) *A write instruction $v \leftarrow v_l$, with global variable $v \in V_s$ and local variable $v_l \in V_l$, is substituted by the following sequence of instructions.*

```
    replace $v \leftarrow v_l$ with
+0      $U(v) \leftarrow_{v\bar{v}}$ TRUE
+1      $v \leftarrow_{v\bar{v}} v_l$
+2      if $\neg Check(U(v))$ or $v \notin U$ then
+3          $Abort$
```

First, v is added to U. Next, the original assignment to v is executed along with a write to $\bar{v}$ (Line +1). Finally, the validity and containment in U is checked.

The variables belonging to output variables have to additionally update their second replicas. We define two rules which are applied *before* Rules 3 and 4.

Rule 5 (Writing variable $v \in V_o$ in blocks $Exec_1$) *The following instruction is inserted before a write instruction $v \leftarrow v_l$ in block $Exec_1$, with local variable $v_l \in V_l$ and output global variable $v \in V_o$.*

```
    before $v \leftarrow v_l$ insert
+1      $\ddot{v} \leftarrow \sim v_l$
```

Rule 5 writes the 1-complement of v_l's value into the replica variable $\ddot{v}$, before a write operation executes in $Exec_1$ stores the value of v_l into v. That guarantees that the message m represented by the variables in V_o is invalid with respect to the variables in $\ddot{V}_o$. In case a fault induces a jump out of the traversal directly to Line 84, the message in V_o can be safely sent since it is invalid.

Rule 6 (Writing variable $v \in V_o$ in blocks $Exec_2$) *The following instruction is inserted before the last write instruction $v \leftarrow v_l$ in block $Exec_2$, with local variable $v_l \in V_l$ and output global variable $v \in V_o$.*

```
    before $v \leftarrow v_l$ insert
+1      $\ddot{v} \leftarrow v_l$
```

Rule 6 writes again into $\ddot{v}$, but this time it writes the value of v_l. If a fault induces a jump out the traversal at this point, the value of v_l is correct by the fault-frequency assumption. Hence, if the message in V_o is valid, then it is the correct message. Note that if v is written multiple times in the block, only the last write to v is accompained by a write to $\ddot{v}$.

4.2.6 Resetting and validation

The final two blocks introduced into p_h are *Reset* and *Validate* (Algorithm 5). Closely resembling Rule 3, the *Reset* block loops over all variables in O and restores their old values, while saving their new values in N. Moreover, *Reset* checks the validity of O before finishing.

The *Validate* block first checks the validity of data structures U and N. Next, it checks that the final value of each variable $v \in V_s$ is the same as the value of $N(v)$. Both **foreach** loops check that every variable in U is also in N and vice versa. Finally, the *Validate* block checks the message validity of the input and output messages. The check at Line 69 in conjunction with the message validity check in the *Filter* block guarantee that the input variables V_i are valid not only with respect to $\bar{V}_i$ but also with respect to $\ddot{V}_i$. The check at Line 71 guarantees that the output variables V_o are valid with respect to $\ddot{V}_o$ if the output message is valid with respect to $\bar{V}_o$.

4.2.7 Control-flow gates

SEI-hardening is straightforward to be proved correct as long as faults do not affect the control flow of the program. Control-flow faults entangle the reasoning because they can change in several different ways the order in which the instructions are executed. To simplify the design of SEI, we introduce control-flow gates, a mechanism that "confines" the control-flow faults within the blocks.

Intuitively, control-flow gates enforce that a fault cannot make the process leave the block where the fault occurs without crashing the process. The scheme mainly provides two guarantees: First, a fault in

```
block Reset
33  foreach $v \in O$ do
34      if $v \notin N$ then
35          if $\neg Check(v)$ then
36              Abort
37          $N(v) \leftarrow_{Cpy} v$
38          if $\neg Check(\mathring{N}(v))$ or $v \in N$ then
39              Abort
40          $\mathring{N}(v) \leftarrow_{v\bar{v}}$ TRUE
41          if $\neg Check(v)$ or $v \neq N(v)$ then
42              Abort
43      $v \leftarrow_{Cpy} O(v)$
44      if $\neg Check(\mathring{N}(v))$ or $v \notin N$ then
45          Abort
46  if $\neg Check(O)$ then
47      Abort
```

```
block Validate
59  if $\neg Check(U)$ or $\neg Check(N)$ then
60      Abort
61  foreach $v \in N$ do
62      if $\neg Check(v)$ or $v \neq N(v)$ then
63          Abort
64      if $v \notin U$ then
65          Abort
66  foreach $v \in U$ do
67      if $v \notin N$ then
68          Abort
69  if $\neg CheckMessage(V_i)$ then
70      Abort
71  if $\neg CheckMessage(V_o)$ then
72      Abort
```

Algorithm 5: *Reset* and *Validate* blocks

```
e1  $B_1$
e2  if $cf$ = TRUE then
e3      Abort
e4  $cf \leftarrow_{v\bar{v}}$ TRUE
e5  $B_2$
e6  if $cf$ = FALSE then
e7      Abort
```

Algorithm 6: Example: single control-flow gate

```
e1  $B_1$                                   (* Block 1 *)
e2  if $cf_1$ = TRUE then                   (* Gate 1 *)
e3      Abort
e4  $cf_1 \leftarrow_{v\bar{v}}$ TRUE
e5  $B_2$                                   (* Block 2 *)
e6  if $cf_2$ = TRUE then                   (* Gate 2 *)
e7      Abort
e8  $cf_2 \leftarrow_{v\bar{v}}$ TRUE
e9  if $cf_1$ = FALSE then
e10     Abort
e11 $B_3$                                   (* Block 3 *)
e12 if $cf_2$ = FALSE then                  (* Gate 3 *)
e13     Abort
```

Algorithm 7: Example: sequence of control-flow gates

the current block cannot jump back into a previous block. Second, a fault in the current block cannot jump into the next block.

To understand how gates achieve these guarantees, consider the example program shown in Algorithm 6 containing two blocks, B_1 and B_2, and a variable cf initialized with the value FALSE (cf stands for control-flow flag). We assume that in fault-free traversals, no instruction in B_1 jumps into B_2, nor any instruction in B_2 jumps into B_1. Blocks can be seen as strictly separate "phases" of a fault-free traversal. In this example, we also assume that cf is not corrupted by a fault, only the control flow, *i.e.*, the program counter.

Assume a control-flow fault occurs in B_2. The fault cannot jump into B_1 without crashing the process. We show why: We know that Line e4 is executed correctly since the fault occurs in B_2, which is after Line e4. A jump from B_2 into B_1 would execute instructions in B_1 and eventually execute Line e2. Since we assume at most one fault per traversal, the second time Line e2 is executed, the process crashes because cf is TRUE. The process does not crash the first time Line e2 is executed because cf is initialized with FALSE (outside the pseudo-code).

Now assume a control-flow fault occurs in B_1. The fault cannot jump into B_2 without crashing the process. A jump from B_1 into B_2 would execute instructions in B_2 and eventually execute Line e6. We know that Line e6 is executed correctly since the fault occurs in B_1, which is before Line e6. The process

```
procedure $FirstGate$
3     if $\neg Check(cf_g)$ or $cf_g =$ TRUE then
4         $Abort$
5     $cf_p, cf_1, cf_r, cf_2, cf_c \leftarrow_{v\bar{v}}$ FALSE
6     if $\neg Check(cf_g)$ or $cf_g =$ TRUE then
7         $Abort$
8     $cf_g \leftarrow_{v\bar{v}}$ TRUE
procedure $Gate(cf_{\text{next}}, cf_{\text{prev}})$
+0    if $\neg Check(cf_{\text{next}})$ or $cf_{\text{next}} =$ TRUE then
+1        $Abort$
+2    $cf_{\text{next}} \leftarrow_{v\bar{v}}$ TRUE
+3    if $\neg Check(cf_{\text{prev}})$ or $cf_{\text{prev}} =$ FALSE then
+4        $Abort$
```

```
procedure $LastGate$
73    $cf_g \leftarrow_{v\bar{v}}$ FALSE
74    if $\neg Check(cf_p)$ or $cf_p =$ FALSE then
75        $Abort$
76    if $\neg Check(cf_1)$ or $cf_1 =$ FALSE then
77        $Abort$
78    if $\neg Check(cf_r)$ or $cf_r =$ FALSE then
79        $Abort$
80    if $\neg Check(cf_2)$ or $cf_2 =$ FALSE then
81        $Abort$
82    if $\neg Check(cf_c)$ or $cf_c =$ FALSE then
83        $Abort$
```

Algorithm 8: Control-flow gates

crashes because Line e4 is skipped by the fault and cf is initially FALSE. In fault-free traversals, the process does not crash since cf is set to TRUE at Line e4.

So far, the scheme partially confines faults within blocks: a fault in B_1 can either jump back inside B_1 or jump exactly at $cf =$ TRUE (Line e4); likewise, a fault in B_2 can either jump back inside B_2 or jump exactly at $cf =$ TRUE. A fault in B_1 or B_2 can still leave the traversal completely jumping to some line *after* Line e7, however. We deal with faults leaving the traversal in Section 4.4.

The scheme described above can be combined to confine faults in several blocks. Algorithm 7 shows an example with 3 blocks. Gates 1 and 2 guarantee that no control-flow fault can jump from B_1 into B_2 and vice versa; whereas Gates 2 and 3 guarantee that no control-flow fault can jump from B_2 into B_3 and vice versa. Additionally, Gates 1, 2 and 3 together guarantee that no fault can jump from B_1 into B_3 because the gates are chained: the check of the assignment to cf_2 (Lines e6 and e8) takes place before the last check of cf_1 (Line e9). If a fault in B_1 jumps into B_3, then cf_2 is FALSE and the process crashes. Recall that only one fault can occur per traversal by assumption, hence, no second fault occurs to jump over Line e12. If a fault in B_1 jumps exactly at Line e8, then cf_2 is set to TRUE, but the process crashes at Line e9 anyhow because cf_1 is FALSE.

The generalization of this scheme is used in SEI-hardening (Algorithm 1) and is described in Algorithm 8. The following control-flow flags are used: cf_g is the control-flow flag of $Prepare_1$, cf_p of $Prepare_2$, cf_1 of $Exec_1$, cf_r of $Reset$, cf_2 of $Exec_2$, and cf_c of $Validate$.

The control-flow flag cf_g additionally protects the whole traversal when a fault jumps out of the traversal by forcing a subsequent traversal to crash the process. Moreover, the hardening of the blocks guarantees that V_o is either invalid with respect to $\ddot{V}$ or it is correct if such a fault occurs.

Definition 45 (First initialization of the control-flow flags) *In the first state of the first traversal, all control-flow flags except cf_g are initialized with* TRUE*; cf_g is initialized with* FALSE.

Remember that the initial state of the first traversal of an execution is correct by assumption (Assumption 6). In each $Gate(\ldots)$, the flags have to be initially FALSE, otherwise the process crashes at Line +0 of the gate. As shown in Algorithm 8, $FirstGate$ initializes all flags with FALSE if and only if cf_g is FALSE.

If one fault jumps out of the current traversal A, cf_g is not set to FALSE since Line 73 is skipped. In a subsequent traversal B, a fault might occur since we assume at most one fault per traversal. If no fault occurs, process π crashes in the check of Line 3 since cf_g is TRUE. A fault could, nevertheless, occur and jump over this line. The double check for $cf_g =$ TRUE at Line 6 guarantees that no fault can skip both checks and still reset the other control-flow flags to FALSE. A fault in traversal B that skips both checks ends up also skipping the reset of the control-flow flags. If process π does not skip the reset of the flags, then it must execute one of the checks of cf_g, which causes it to crash.

If some flag other than cf_g is initially TRUE in traversal B and not reset at Line 5, then π will crash in the respective gate at Line +0. The only way to set the cf_g to FALSE is by executing the $LastGate$. However, $LastGate$ crashes the process if any of the other control-flow flags is FALSE. As we will show in the correctness proof of Section 4.4, the only possible scenario is a fault in the subsequent traversal B

Set	Description	Relation
V	all variables of π	
V_g	global variables of π	$V_g \subset V$
V_l	local variables of π	$V_l \subset V$
V_r	replica variables of π	$V_r \subset V$
V_p	variables used by program p	$V_p \subset V$
V_s	global variables used by program p	$V_s = V_p \cap V_g$
V_h	global variables used by hardening	$V_h \subset V_g$ and $V_h \cap V_s = \{\ \}$
V_i	input message variables	$V_i \subset V_s$
$\bar{V}_i$	input message's first replica	$\bar{V}_i \subset V_r$
$\ddot{V}_i$	input message's second replica	$\ddot{V}_i \subset V_g$ or $\ddot{V}_i \subset V_l$
V_o	output message variables	$V_o \subset V_s$
$\bar{V}_o$	output message's first replica	$\bar{V}_o \subset V_r$
$\ddot{V}_i$	output message's second replica	$\ddot{V}_o \subset V_g$ or $\ddot{V}_o \subset V_l$
O	variables with old values of variables in V_s	$O \subset V_h$
$\mathring{O}$	variables marking modified variables in V_s	$\mathring{O} \subset V_h$
N	variables with new values of variables in V_s	$N \subset V_h$
$\mathring{N}$	variables marking modified variables in V_s	$\mathring{N} \subset V_h$
U	variables marking modified variables in V_s ($Exec_2$)	$U \subset V_h$
$Flags$	control-flow variables of gates	$Flags \subset V_h$

Table 2: Summary of all variable sets used in SEI-hardening

jumping over the *FirstGate* into the same block K where the previous traversal A was left by the first fault. We show that such sequence of faults is equivalent to a single fault confined to block K.

4.3 Correctness with block-confined faults

Our proof is divided in two parts. In this section, we only consider the *blocks* of SEI-hardening (Algorithm 1) and assume faults cannot escape them. In Section 4.4, we prove that the *gates* can guarantee this assumption. We define block-confined faults as follows.

Assumption 7 (Block-confined faults) *A fault corrupting the control flow –* i.e.*, the program counter – does not set the pc to an instruction other than from the block where the fault occurs.*

Given Assumption 7, we show that if the following invariant holds at the beginning of a traversal, then it holds at the end of the traversal. That is technically not an inductive invariant over the steps of the system, but rather over the complete traversals, hence, we call this invariant a *traversal invariant.* The traversal invariant asserts that if a variable v is corrupt at state s (*i.e.*, different from there reference value), then v is invalid at s. In this section, the traversal invariant is assumed to hold in the initial state of the traversal s_b. We prove that the traversal invariant holds in the final state of the traversal s_e.

Property 5 (Traversal invariant) *Given a state s of a traversal,*

(a) for all $v \in V_g \backslash (V_i \cup V_o)$, if $s[v] \neq r[v]$, then $s[v] \neq s[\bar{v}]$;

(b) for all $v \in V_i$, if $s[v] \neq r[v]$, then $s[v] \neq s[\bar{v}]$ or $s[v] \neq s[\ddot{v}]$; and

(c) for all $v \in V_o$, if $s[v] \neq r[v]$, then $s[v] \neq s[\ddot{v}]$.

Particularly important is the distinction between the cases of general global variables and input/output variables. Global variables other than V_i and V_o are protected by their replica state between traversals and by the SEI-hardening algorithms during traversals. If some fault occurs until the beginning of the traversal and corrupts a variable v, then $s_b[v] \neq s_b[\bar{v}]$ by corruption coverage. We will show that if some variable v is corrupted during the traversal, then $s_e[v] \neq s_e[\bar{v}]$.

In contrast, variables in V_i might arrive already corrupted from the network. If no fault occurs until the beginning of the traversal, then it holds that $s_b[v] = s_b[\bar{v}]$ for all $v \in V_i$ and $s_b[v] \neq s_b[\ddot{v}]$ for some

State	Description	Line
s_b	the state at the first line of program p_h (*Filter*)	1
s_g	the state at the first line of $\textit{Prepare}_1$	9
s_p	the state at the first line of $\textit{Prepare}_2$	18
s_{x_1}	the state at the first line of $\textit{Exec}_1$	27
s_r	the state at the first line of *Reset*	33
s_{x_2}	the state at the first line of $\textit{Exec}_2$	53
s_c	the state at the first line of *Validate*	59
s_e	the state immediately after the last line of program p_h	84
r_b	the reference state at the first line of program p_h	1
r_e	the reference state immediately after the last line of program p_h	84

Table 3: States used in the correctness proofs

$v \in V_i$. In other words, the input message is valid with respect to $\bar{V}_i$, but invalid with respect to $\ddot{V}$. Moreover, if some fault occurs until the beginning of the traversal corrupting a variable $v \in V_i$, then $s_b[v] \neq s_b[\bar{v}]$ by corruption coverage.

Finally, the output variables have a stronger requirement: if some variable $v \in V_o$ is corrupted during the traversal, then it should hold that $s_e[v] \neq s_e[\ddot{v}]$. By definition, the values of variables v and $\ddot{v}$ are sent over the network, but not of variable $\bar{v}$ – remember that an implementation using hardware error-detection codes cannot access the ECC memory to retrieve the value of $\bar{v}$. Since any check after the traversal but before the sending could be skipped by a fault we cannot rely on $\bar{v}$ for guaranteeing error isolation – remember that we do not make any fault frequency assumption outside the traversals. The traversal invariant has to make an assertion on the variables that are effectively sent over the network, hence, it must hold that v and $\ddot{v}$ have the same value at s_e.

4.3.1 Conventions

Since we model here high-level operations of the algorithms, each line of the algorithms and Rules 1-6 are in fact a sequence of instructions, *i.e.*, a sequence of *Next* steps. Remember that we *do not* consider all instructions of a line to be executed atomically with respect to *Fault* steps. We use the following convention to refer to a state:

The state s at Line X is the state s before the execution of the first instruction of the sequence of instructions represented by Line X.

The state s immediately after Line X is the state s at Line X+1.

A fault occurs or takes place at state s: s is the state before the fault transition is taken and s' is the state after the fault has taken place.

Since in this section a fault is always confined to the block where it occurs, the state at the first line of every block exists. Moreover, the state after the last line (Line 84) exists. We use these states to guide our proofs; they are listed in Table 3. Note that if a traversal never completes, it can be equated to a crash so we do not need to consider it.

In a traversal, states are totally ordered starting from state s_b until state is s_e. We use the $\prec$ relation to indicate that a state precedes another.

Faults are one way to corrupt variables, the other way is via error propagation occurring in a *Next* state transition, *i.e.*, the execution of instructions over corrupt source operands might result in corrupt target operands. We use the expressions *modified by a fault* or *modified by an assignment* to mean that a variable was modified at a state s by either a *Fault* transition or by a *Next* transition, respectively. Sometimes we have to differentiate these two cases referring to the last modification of a variable up to the current state s. For that we define what it means for a variable to be *determined by a fault* and *determined by an assignment* as follows.

Definition 46 (Determined by a fault at state s) *A variable $v \in V$ is determined by a fault at state s if and only if the last fault before s occurs at a state $s_f \prec s$, takes place at state s'_f, and $s_f[v] \neq s'_f[v] = s[v]$.*

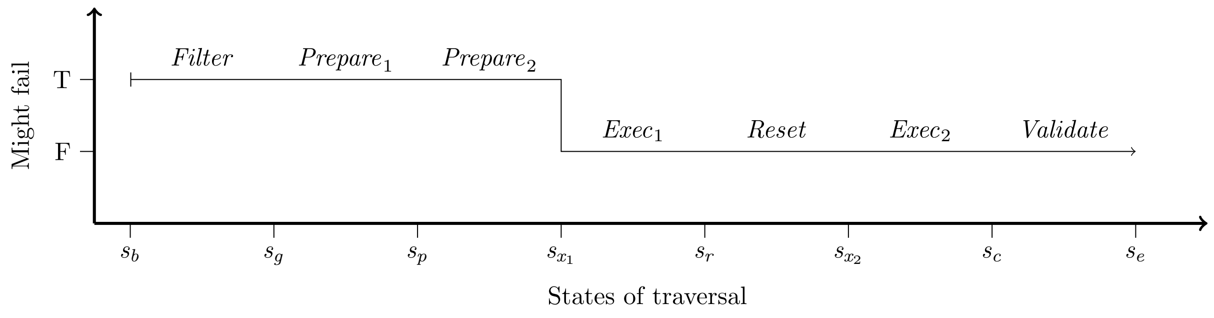


Figure 5: Fault assumption for Lemmas 2 and 3

Definition 47 (Determined by an assignment at state s) *A variable $v \in V$ is determined by an assignment i at state s if and only if v is not determined by a fault at s, i is the last assignment to v performed at state $s_a \prec s$, i transforms state s_a into s'_a, and $s'_a[v] = s[v]$.*

In our proofs, remember that we implement the containment of a variable v in a set data structure – *e.g.*, $v \in O$ – with an auxiliary map data structure (see Definition 42). When a variable v is contained by a set data structure D, then its value in the auxiliary map data structure $\mathring{D}$ is TRUE – *e.g.*, $s[\mathring{O}(v)] = \text{TRUE}$.

4.3.2 Faults in the *Initialization* blocks

We start by showing that, if a fault occurs in one of the *Initialization* blocks and process π does not crash, then at the beginning of *Exec*$_1$, all the variables in V_i can be checked against $\bar{V}_i$. Moreover, all data structures are either correctly initialized or invalid. See Figure 5 for our fault assumption in the following lemmas.

Lemma 2 asserts that if the input message has arrived invalid from the network, then it is either discarded or the trust from the $\ddot{V}_i$ is transferred to $\bar{V}_i$ – even if a fault occurs in one of the *Initialization* blocks. In other words, the variables of the input message are invalid with respect to $\bar{V}_i$ before *Exec*$_1$ starts if they are invalid with respect to $\ddot{V}_i$ when the traversal starts. Again, the trust transfer allows *Exec*$_1$ and the following blocks to treat variables in V_i as ordinary program-state variables – leveraging cheap hardware checks if these are available. Moreover, the lemma asserts that the bookkeeping data structures O, N and U are all correctly initialized even if a fault occurs in the *Initialization* blocks.

Lemma 2 *Assume the traversal invariant holds at s_b, a fault occurs in Filter, Prepare$_1$, or Prepare$_2$, and π does not crash. For all $v \in V_i$, if $s_b[v] \neq s_b[\ddot{v}]$, then v is invalid at s_{x_1}. Moreover, for all $v \in V_s$,*

- *$s_{x_1}[\mathring{O}(v)] = \text{FALSE}$ or $\mathring{O}(v)$ is invalid at s_{x_1},*
- *$s_{x_1}[\mathring{N}(v)] = \text{FALSE}$ or $\mathring{N}(v)$ is invalid at s_{x_1},*
- *and $s_{x_1}[U(v)] = \text{FALSE}$ or $U(v)$ is invalid at s_{x_1}.*

Proof:
1. CASE: A fault occurs in *Filter*
   1.1. *Prepare*$_1$, *Prepare*$_2$, and *Validate* execute correctly since faults are block-confined by Assumption 7.
   1.2. No variable in V_i is modified by an instruction between s_b and s_e by Assumption 5.
   1.3. For all $v \in V_i$, $s_{x_1}[v] = s_{x_1}[\ddot{v}]$ and $s_{x_1}[v] = s_{x_1}[\bar{v}]$, otherwise π crashes in *Validate* (Line 69) by Steps 1.1 and 1.2.
   1.4. For all $v \in V_i$, $s_{x_1}[v] = s_b[v]$ and $s_{x_1}[\ddot{v}] = s_b[\ddot{v}]$ by Steps 1.2 and 1.3 and by corruption coverage.
   1.5. For all $v \in V_i$, $s_b[v] = s_b[\ddot{v}]$ by Step 1.4.
   1.6. For all $v \in V_s$, $s_{x_1}[\mathring{O}(v)] = \text{FALSE}$, $s_{x_1}[\mathring{N}(v)] = \text{FALSE}$, and $s_{x_1}[U(v)] = \text{FALSE}$ since *Prepare*$_2$ executes correctly by Step 1.1.
2. CASE: A fault occurs in *Prepare*$_1$
   2.1. *Filter*, *Prepare*$_2$, and *Validate* execute correctly since faults are block-confined by Assumption 7.
   2.2. No variable in V_i is modified by an instruction between s_b and s_e by Assumption 5.
   2.3. For all $v \in V_i$, $s_{x_1}[v] = s_{x_1}[\ddot{v}]$ and $s_{x_1}[v] = s_{x_1}[\bar{v}]$, otherwise s_{x_1} does not exist since *Filter* terminates the traversal at Line 2 or π crashes in *Validate* (Line 69) by Steps 2.1 and 2.2.

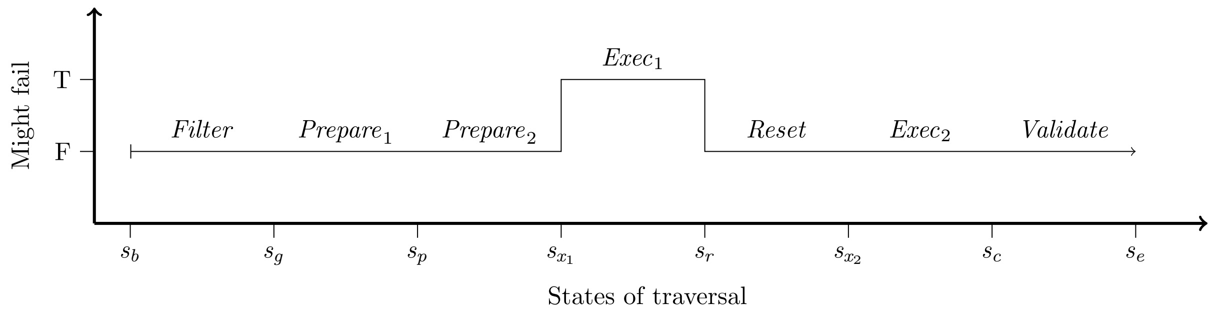


Figure 6: Fault assumption for Lemmas 4 and 5

2.4. For all $v \in V_i$, $s_{x_1}[v] = s_b[v]$ and $s_{x_1}[\ddot{v}] = s_b[\ddot{v}]$ by Steps 2.2 and 2.3 and by corruption coverage.
2.5. For all $v \in V_i$, $s_b[v] = s_b[\ddot{v}]$ by Step 2.4.
2.6. For all $v \in V_s$, $s_{x_1}[\mathring{O}(v)] =$ FALSE, $s_{x_1}[\mathring{N}(v)] =$ FALSE, and $s_{x_1}[U(v)] =$ FALSE since *Prepare*$_2$ executes correctly by Step 2.1.

3. CASE: A fault occurs in *Prepare*$_2$
3.1. *Filter* and *Prepare*$_1$ execute correctly since faults are block-confined by Assumption 7.
3.2. No variable in V_i is modified by an instruction between s_b and s_p by Assumption 5.
3.3. For all $v \in V_i$, $s_p[v] = s_p[\ddot{v}]$ and $s_p[v] = s_p[\bar{v}]$, otherwise s_p does not exist since *Filter* terminates the traversal at Line 2 by Steps 3.1 and 3.2.
3.4. For all $v \in V_i$, $s_p[v] = s_b[v]$ and $s_p[\ddot{v}] = s_b[\ddot{v}]$ by Steps 3.2 and 3.3 and by corruption coverage.
3.5. For all $v \in V_i$, $s_b[v] = s_b[\ddot{v}]$ by Step 3.4.
3.6. For all $v \in V_s$, $s_p[\mathring{O}(v)] =$ FALSE, $s_p[\mathring{N}(v)] =$ FALSE, and $s_p[U(v)] =$ FALSE by correct execution of *Prepare*$_1$.
3.7. For all $v \in V_i$, if $s_{x_1}[v] \neq s_p[v]$, then v is invalid at s_{x_1} by corruption coverage.
3.8. For all $v \in V_s$, if $s_{x_1}[\mathring{O}(v)] \neq$ FALSE, then $\mathring{O}(v)$ is invalid at s_{x_1} by corruption coverage.
3.9. For all $v \in V_s$, if $s_{x_1}[\mathring{N}(v)] \neq$ FALSE, then $\mathring{N}(v)$ is invalid at s_{x_1} by corruption coverage.
3.10. For all $v \in V_s$, if $s_{x_1}[U(v)] \neq$ FALSE, then $U(v)$ is invalid at s_{x_1} by corruption coverage.

□

We now show that if *Exec*$_1$, *Reset* and *Exec*$_2$ correctly execute on correct variables, then the result of the traversal is correct (or invalid).

Lemma 3 *Assume the traversal invariant holds at s_b, no fault occurs in Exec$_1$, Reset and Exec$_2$, and π does not crash. For every variable $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.*

Proof:
1. *Filter*, *Prepare*$_1$ and *Prepare*$_2$ do not write into any $v \in V_s$ by definition (Algorithm 2).
2. For all $v \in V_s$, either $s_{x_1}[v] = s_b[v]$ by Step 1, or v is invalid at s_{x_1} by corruption coverage if a fault modified v.
3. For all $v \in V_s$ read in *Exec*$_1$ for the first time at some state s, v is valid at s since π does not crash in the check of Line +1, Rule 2.
4. For all $v \in V_s$ read in *Exec*$_1$ for the first time at some state s, $s[v] = s_{x_1}[v] = s_b[v] = r_b[v]$ by Steps 1, 2 and 3 and traversal invariant.
5. For all $v \in V_s$, $v \notin O$, $v \notin N$ and $v \notin U$ at state s_{x_1} by Lemma 2, and $\mathring{O}(v)$, $\mathring{N}(v)$ and $U(v)$ are valid at s_{x_1} since no fault occurs after *Prepare*$_2$ and since π does not crash at the checks of Lines 46 and 59.
6. *Exec*$_1$, *Reset*, and *Exec*$_2$ execute correctly (by assumption) on correct variables by Steps 3, 4 and 5 and traversal invariant.
7. For every variable $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e by Step 6, corruption coverage and traversal invariant. □

4.3.3 Faults in the *Exec*$_1$ block

Following the sequence of blocks, we now assume a fault occurs in *Exec*$_1$ (see Figure 6). We first show that if some variable v is determined by an assignment in block *Exec*$_1$, then all instructions in Rule 3 are executed. In other words, if v is modified by an assignment in *Exec*$_1$, $O(v)$ contains v's old value at state s_r. Moreover, v is valid at the initial state s_b. This ensures that *Reset* can correctly rollback state updates performed in $Exec_1$.

Lemma 4 *Assume no faults occurs in Initialization blocks and Reset block, and π does not crash. For any $v \in V_s$ modified by an assignment in block $Exec_1$, it holds that $s_r[O(v)] = s_b[v]$, $s_r[\mathring{O}(v)] = \text{TRUE}$, $O(v)$ and $\mathring{O}(v)$ are valid at s_r, and v is valid at s_b.*

Proof:
1. For all $v \in V_s$, $v \notin O$, $v \notin N$, and $v \notin U$ at state s_{x_1} by assumption.
2. Let some $v \in V_s$ be determined by assignment for the first time in $Exec_1$ at state s_1 immediately after Line +9, Rule 3.
3. $s_1 \prec s_r$ by Assumption 7.
4. $O(v)$ and $\mathring{O}(v)$ are valid at s_r, otherwise π would crash in *Reset*, Line 46.
5. $s_1[\mathring{O}(v)] = \text{TRUE}$, *i.e.*, $v \in O$ at s_1.
   5.1. If no fault occurs before s_1, the value of v is added to O (Line +6, Rule 3) before the value v is modified.
   5.2. If no fault occurs after s_1, the absence of an entry for v would result in a crash right after s_1 due to the check $v \in O$ (Line +11, Rule 3); a contradiction.
6. There is a state $s_2 \prec s_1$ immediately after Line +6, Rule 3, when $\mathring{O}(v)$ is assigned for the first time in $Exec_1$.
   6.1. Assume by contradiction that $s_1 \prec s_2$.
   6.2. $v \notin O$ at state s_1 since *Initialization* blocks execute correctly by assumption.
   6.3. A fault must have skipped Lines +3 to +6 before Line +9 is executed, otherwise $s_2 \prec s_1$.
   6.4. π crashes at Line +11 by Step 6.2 and since no faults occurs after s_1 by Step 6.3; a contradiction.
7. $s_2[O(v)] = s_b[v]$ and v is valid at s_b if no fault occurs before s_2.
   7.1. $s_2[O(v)] = s_b[v]$ since no faults occur before s_2.
   7.2. v is not assigned before s_2 by definition of s_1.
   7.3. π does not crash in the check of Line +1, Rule 3, by assumption.
   7.4. v is valid at s_b by Steps 7.2 and 7.3.
8. $s_2[O(v)] = s_b[v]$ and v is valid at s_b if no fault occurs after s_2.
   8.1. $s_2[O(v)] = s_2[v]$ and v is valid at s_2, otherwise π crashes after s_2 at Line +8, Rule 3.
   8.2. v is not modified by a fault before s_2 since v is valid at s_2 by Step 8.1.
   8.3. v is not modified by an assignment before s_2 by definition of s_1.
   8.4. $s_2[v] = s_b[v]$ by Steps 8.2 and 8.3.
   8.5. $s_2[O(v)] = s_b[v]$ and v is valid at s_b by Steps 8.1 and 8.4.
9. $s_r[O(v)] = s_2[O(v)]$, $s_r[\mathring{O}(v)] = \text{TRUE}$, $O(v)$ is valid at s_r and $\mathring{O}(v)$ is valid at s_r.
   9.1. No instruction writes FALSE into $\mathring{O}(v)$ after s_2 by definition.
   9.2. $s_r[\mathring{O}(v)] = \text{TRUE}$ and $\mathring{O}(v)$ is valid at s_r by Steps 6, 9.1, 4 and 5.
   9.3. $O(v)$ is not modified by assignment after s_1 if $\mathring{O}(v)$ is not modified by a fault after s_1.
      9.3.1. Assume there is a state s_3 at Line +3 of Rule 3 and $s_3 \succ s_1$.
      9.3.2. $s_3[\mathring{O}(v)] = \text{TRUE}$ since $\mathring{O}(v)$ is not modified by a fault by assumption, since $s_2 \prec s_1 \prec s_3$ and by Steps 9.1 and 5.
      9.3.3. π crashes at Line +5 by Step 9.3.2; a contradiction.
   9.4. $O(v)$ is not modified by assignment after s_1 if $\mathring{O}(v)$ is modified by a fault after s_1.
      9.4.1. Assume there is a state s_3 at Line +3 of Rule 3 and $s_3 \succ s_1$.
      9.4.2. $\mathring{O}(v)$ is invalid by corruption coverage and since $s_2 \prec s_1 \prec s_3$.
      9.4.3. π crashes at Line +5 by Step 9.4.2; a contradiction.
   9.5. $s_r[O(v)] = s_2[O(v)]$ by Steps 9.3 and 9.4, and $s_r[\mathring{O}(v)] = \text{TRUE}$ by Step 9.2.
10. $s_r[O(v)] = s_b[v]$, $s_r[\mathring{O}(v)] = \text{TRUE}$ and v is valid at s_b by Steps 4, 7, 8 and 9. □

Using Lemma 4, we show next that for all $v \in V_s$, v either has the expected result at s_e, or v is invalid.

Lemma 5 *Assume the traversal invariant holds at s_b, a fault occurs in $Exec_1$, and π does not crash. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.*

Proof:
1. If v is determined by a fault before s_e, we are done by corruption coverage.
2. *Initialization*, *Reset*, $Exec_2$, and *Validate* execute correctly by Assumption 7.
3. For all $v \in V_s$, $s_{x_2}[v] = r_b[v]$ or v is invalid at s_{x_2}.

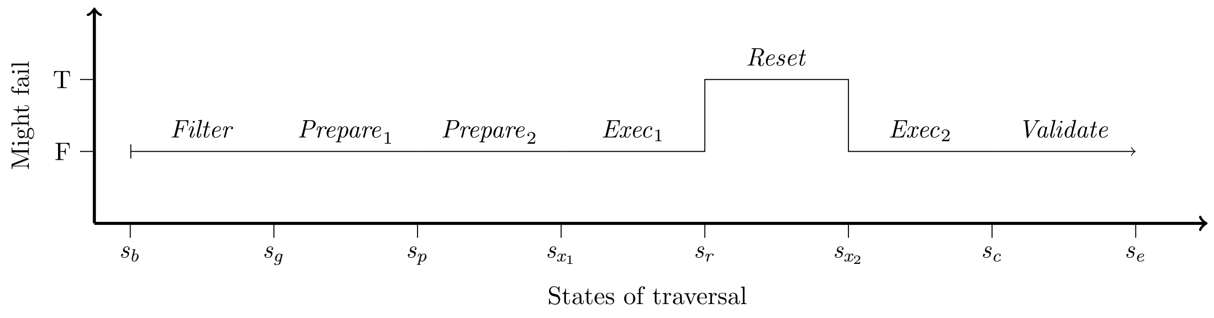


Figure 7: Fault assumption for Lemmas 6, 7 and 8

3.1. For all $v \in V_s$ assigned in *Exec*$_1$, v is valid at s_b by Lemma 4.
3.2. For all $v \in V_s$ assigned in *Exec*$_1$, $s_r[O(v)] = s_b[v]$ by Lemma 4.
3.3. For all $v \in V_s$ assigned in *Exec*$_1$, $s_b[v] = r_b[v]$ by Step 3.1 and by traversal invariant.
3.4. For all $v \in V_s$ assigned in *Exec*$_1$, $s_{x_2}[v] = s_r[O(v)] = r_b[v]$ by Steps 3.2, 3.3 and 2.
3.5. For all $v \in V_s$, $s_{x_2}[v] = r_b[v]$ or v is invalid at s_{x_2} by Step 3.4, traversal invariant, corruption coverage, and since no other block writes to $v \in V_s$ before *Reset* except *Exec*$_1$ by definition.

4. For all $v \in V_s$ read in *Exec*$_2$, v is valid at s_{x_2}, otherwise π would crash at Line +1, Rule 2, by Step 2.
5. For all $v \in V_s$ modified in *Exec*$_2$, $s_c[v] = r_e[v]$, since *Exec*$_2$ correctly executes on correct inputs by Steps 2, 3, and 4.
6. For all $v \in V_s$ modified in *Exec*$_2$, $s_e[v] = s_c[v] = r_e[v]$ by Step 2, 5 and since no other block writes to $v \in V_s$ after *Exec*$_2$ by definition.
7. For all $v \in V_s$, $s_e[v] = r_e[v]$ or v is invalid at s_e by Step 5, traversal invariant, corruption coverage since no other block writes to $v \in V_s$ after *Exec*$_2$ by definition. $\square$

4.3.4 Faults in the *Reset* block

Our next three lemmas assume that a fault occurs in the *Reset* block (see Figure 7). We start by showing that the state at the beginning of *Reset* is the expected final state, and any variable modified in *Exec*$_1$ has its old value correctly stored in O.

Lemma 6 *Assume the traversal invariant holds at s_b, no faults occurs in Initialization blocks and in Exec$_1$, and π does not crash. For all $v \in V_s$, it holds that $s_r[v] = r_e[v]$ or v is invalid at s_r.*

Proof:
1. For all $v \in V_s$ read for the first time at some state s during the execution of *Exec*$_1$, it holds that $s[v] = r_b[v]$.
   1.1. v is valid since π does not crash by assumption, every read is preceded by a validity check by Rule 2, and no fault occurs between s_b and s_r.
   1.2. $s[v] = r_b[v]$ by Step 1.1, traversal invariant and Lemma 2.
2. For all $v \in V_s$ read or modified in *Exec*$_1$, $s_r[v] = r_e[v]$ since no fault occurs between s_b and s_r by assumption and π reads correct inputs by Step 1.
3. For all $v \in V_s$, $s_r[v] = r_e[v]$ or v is invalid at s_r by Step 2 and traversal invariant. $\square$

We now show that if some $v \in N$ at the beginning of *Exec*$_2$, then $N(v)$ holds the reference final value of v.

Lemma 7 *Assume the traversal invariant holds at s_b, a fault occurs in Reset, and π does not crash. For all $v \in V_s$, if $v \in N$ at s_{x_2}, then $s_{x_2}[N(v)] = r_e[v]$ and $N(v)$ and $\mathring{N}(v)$ are valid at s_{x_2}.*

Proof:
1. *Initialization*, *Exec*$_1$, *Exec*$_2$, and *Validate* execute correctly by assumption.
2. $N(v)$ and $\mathring{N}(v)$ are valid at s_{x_2}, otherwise π crashes in *Validate*.
3. There is a state s_2 at Line 37 when $\mathring{N}(v)$ is assigned TRUE for the first time in *Reset*.
   3.1. $\mathring{N}(v)$ is set to FALSE in *Initialization* blocks, which execute correctly by assumption.
   3.2. $s_{x_2}[\mathring{N}(v)] =$ TRUE, *i.e.*, $v \in N$ at s_{x_2}, by assumption.
   3.3. Only *Reset* block writes TRUE to $\mathring{N}(v)$ by definition.
4. $s_2[N(v)] = s_r[v]$ and v is valid at s_r if no fault occurs before s_2.

4.1. $s_2[N(v)] = s_r[v]$ since no faults occur before s_2.
4.2. v is not modified by assignment after s_r and before s_2 by definition of s_2.
4.3. π does not crash in the check of Line 35 by assumption.
4.4. v is valid at s_r by Steps 4.2 and 4.3.
5. $s_2[N(v)] = s_r[v]$ and v is valid at s_r if no fault occurs after s_2.
5.1. $s_2[N(v)] = s_2[v]$ and v is valid at s_2, otherwise π crashes at Line 41.
5.2. v is not determined by a fault before s_2 since v is valid at s_2 by Step 5.1.
5.3. v is not modified by an assignment after s_r and before s_2.
5.3.1. Assume there is a state s_1 such that $s_r \prec s_1 \prec s_2$ and s_1 is the state after Line 43.
5.3.2. If no fault occurs before s_1, then $s_1[\mathring{N}(v)] = \text{TRUE}$ by correct execution up to s_1 and Step 1.
5.3.3. If no fault occurs before s_1, there is a state $s_3 \prec s_1$ at Line 40 by Step 5.3.2; a contradiction with definition of s_2 (first time v is assigned in *Reset*).
5.3.4. If no fault occurs after s_1, π crashes in the check of Line 44.
5.4. $s_2[v] = s_r[v]$ by Steps 5.2 and 5.3.
5.5. $s_2[N(v)] = s_r[v]$ and v is valid at s_r by Steps 5.1 and 5.4.
6. $s_{x_2}[N(v)] = s_2[N(v)]$ and $s_{x_2}[\mathring{N}(v)] = \text{TRUE}$.
6.1. No instruction writes FALSE into $\mathring{N}(v)$ after s_2 by definition (Algorithm 1) and block assumption.
6.2. $s_{x_2}[\mathring{N}(v)] = \text{TRUE}$ by Steps 6.1, 1 and 2 and definition of s_2.
6.3. $N(v)$ is not modified by assignment after s_2 if $\mathring{N}(v)$ is not modified by a fault after s_2.
6.3.1. Assume there is a state s_3 at Line 37 and $s_3 \succ s_2$.
6.3.2. $s_3[\mathring{N}(v)] = \text{TRUE}$ since $\mathring{N}(v)$ is not modified by a fault by assumption, and since $s_2 \prec s_3$ and by Steps 6.1 and 2.
6.3.3. π crashes at Line 38 by Step 6.3.2; a contradiction.
6.4. $N(v)$ is not modified by assignment after s_2 if $\mathring{N}(v)$ is modified by a fault after s_2.
6.4.1. Assume there is a state s_3 at Line 37 and $s_3 \succ s_2$.
6.4.2. $\mathring{N}(v)$ is invalid at s_3 by corruption coverage and since $s_2 \prec s_3$.
6.4.3. π crashes at Line 38 by Step 6.4.2; a contradiction.
6.5. $s_{x_2}[N(v)] = s_2[N(v)]$ by Steps 6.3 and 6.4, and $s_{x_2}[\mathring{N}(v)] = \text{TRUE}$ by Step 6.2.
7. $s_{x_2}[N(v)] = s_r[v]$, $s_{x_2}[\mathring{N}(v)] = \text{TRUE}$ and v is valid at s_r by Steps 4, 5 and 6.
8. $s_{x_2}[N(v)] = r_e[v]$ by Lemma 6. □

Finally, we show that if a fault occurs in *Reset*, the state at the end of the traversal is the expected state or invalid. There are mainly two cases can happen. First, if a variable v is modified in $Exec_2$, then it must contain the expected final value, otherwise π crashes when comparing U with N (Line 59) or the current value of v with $N(v)$ (Line 62). Second, if a variable v is not modified in $Exec_2$, then it contains either the initial value if it was not modified in $Exec_1$, or it contains the expected final value if it was modified in $Exec_1$. A third case, where v is only modified in *Reset*, is not possible since π would crash when comparing U with N (Line 59).

Lemma 8 *Assume the traversal invariant holds at s_b, a fault occurs in Reset, and π does not crash. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.*

Proof:
1. *Initialization*, $Exec_1$, $Exec_2$ and *Validate* execute correctly by assumption.
2. For all $v \in V_s$ determined by a fault at state s_e, we are done by corruption coverage.
3. For all $v \in V_s$ not modified by assignment in $Exec_1$, *Reset*, or $Exec_2$, we are done by traversal invariant.
4. If suffices to show that for all $v \in V_s$ determined by an assignment in $Exec_1$, *Reset*, or $Exec_2$, if $s_c[v] \neq r_e[v]$, then v is invalid at s_c.
4.1. No $v \in V_s$ is assigned in *Validate* by definition.
4.2. For all $v \in V_s$ determined by assignment in $Exec_1$, *Reset*, or $Exec_2$, $s_e[v] = s_c[v]$ or v is invalid at s_e by Step 4.1 and since no fault occurs in *Validate*.
5. Let $v \in V_s$ be determined by assignment in $Exec_1$, *Reset*, or $Exec_2$. We show that if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.
6. Case: $\mathring{N}(v)$ is invalid at s_{x_2}.
6.1. π crashes in the *Validate* block, Line 59.
7. Case: $s_c[\mathring{N}(v)] = \text{TRUE}$ and $\mathring{N}(v)$ is valid at s_c.
7.1. $v \in U$ at s_c, otherwise π crashes in check of Line 64.
7.2. $s_c[v] = s_{x_2}[N(v)]$, otherwise π crashes in check of Line 62.

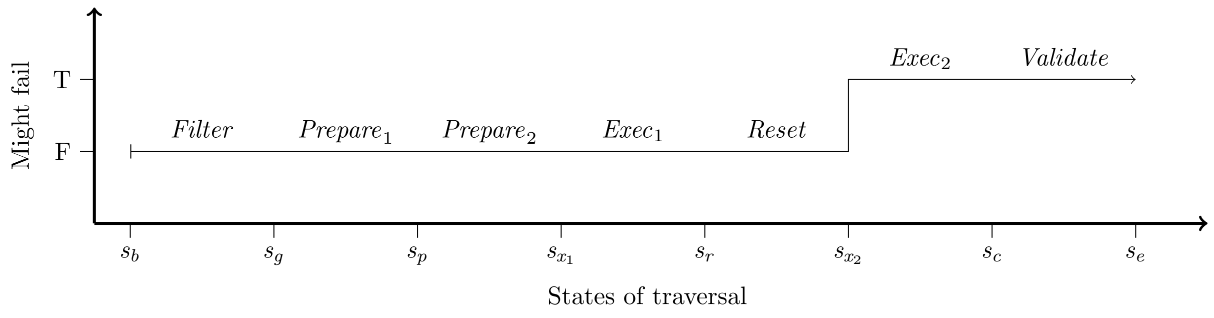


Figure 8: Fault assumption for Lemmas 9 and 10

7.3. $s_c[v] = s_{x_2}[N(v)] = s_r[v] = r_e[v]$ by Lemma 7 and Step 1.
7.4. v is valid at s_c, otherwise π crashes in check of Line 62.
8. CASE: $s_c[\mathring{N}(v)] = \text{FALSE}$ and $\mathring{N}(v)$ is valid at s_c.
8.1. $v \notin U$ at s_c, otherwise π crashes in check of Line 67.
8.2. v is not assigned in *Exec*$_2$ by Steps 1 and 8.1.
8.3. v is not assigned in *Reset*, otherwise $v \in N$ at s_c by Lemma 7.
8.4. v is determined by assignment in *Exec*$_1$ by Steps 5, 8.2 and 8.3.
8.5. $s_r[v] = r_e[v]$ and v is valid at s_r by Step 8.4 and Lemma 6.
8.6. $s_c[v] = s_r[v]$ and v is valid at s_c by Steps 8.2 and 8.3.
8.7. $s_c[v] = r_e[v]$ and v is valid at s_c by Steps 8.5 and 8.6.
9. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e by Steps 2, 3, 4, 6, 7 and 8. □

4.3.5 Faults in the *Exec*$_2$ and *Validate* blocks

We now assume a fault occurs either in the *Exec*$_2$ or in the *Validate* block (see Figure 8). Since most of the blocks execute correctly, proving the correctness of *Exec*$_2$, as well as of *Validate* below, is much simpler than the previous blocks.

Lemma 9 *Assume the traversal invariant holds at s_b, a fault occurs in Exec$_2$, and π does not crash. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.*

Proof:
1. For all $v \in V_s$, if v is determined by a fault, we are done by corruption coverage.
2. *Initialization*, *Exec*$_1$, *Reset*, and *Validate* execute correctly by assumption.
3. For all $v \in V_s$, $s_{x_2}[v] = r_b[v]$ or v is invalid at s_{x_2} by Step 2.
4. Let $v \in V_s$ be such that $s_e[v] \neq r_e[v]$ and v is valid at s_e. We show that if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.
5. CASE: v was assigned in *Exec*$_1$.
5.1. $v \in N$ at $\mathring{s}_c$ by Step 2.
5.2. $N(v)$ and $\mathring{N}(v)$ are valid at s_c, otherwise π crashes in the check of Line 59.
5.3. $v \in U$ at s_c, otherwise π crashes in the check of Line 64.
5.4. $s_c[v] = s_c[N(v)]$, otherwise π crashes in the check of Line 62.
5.5. $s_c[N(v)] = r_e[v]$ by Steps 2 and 5.2.
5.6. $s_e[v] = s_c[v] = r_e[v]$ by Steps 5.4, 5.5 and 2; a contradiction with Step 4.
6. CASE: v was not assigned in *Exec*$_1$ and not assigned in *Exec*$_2$.
6.1. $v \notin O$ at s_r by Step 2.
6.2. v is not assigned in *Reset* by Step 6.1.
6.3. $s_e[v] = r_e[v]$ if v is not determined by a fault since $r_b[v] = r_e[v]$.
6.4. $s_e[v] = r_e[v]$ or v is invalid at s_e by Step 6.3 and 1; a contradiction with Step 4.
7. CASE: v was not assigned in *Exec*$_1$, but was assigned in *Exec*$_2$.
7.1. $v \notin O$ at s_r by Step 2.
7.2. v is not assigned in *Reset* by Step 7.1.
7.3. $s_r[v] = r_b[v]$ or invalid by Step 7.2 and traversal invariant.
7.4. Let s be the state immediately after v is assigned in *Exec*$_2$ (Line +1, Rule 4).
7.5. If no fault occurs before s, then $v \in U$ at s_c by correct exectuion before s and $U(v)$ is valid at s_c, otherwise π crashes in the check of Line 59.

7.6. If no fault occurs after s, then $v \in U$ at s_c and $U(v)$ is valid at s_c, otherwise π crashes in the check of Line +2, Rule 4.
7.7. π crashes at Line 67; a contradiction.
8. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e by Steps 1, 5, 6 and 7. □

Finally, we assume that a fault occurs in the *Validate* block and show that the traversal invariant holds after the traversal has finished.

Lemma 10 *Assume the traversal invariant holds at s_b, a fault occurs in Validate, and π does not crash. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e.*

Proof:
1. For all $v \in V_s$, if v is determined by a fault, we are done by corruption coverage.
2. *Validate* does not modify any $v \in V_s$ by definition.
3. *Initialization*, *Exec*$_1$, *Reset*, and *Exec*$_2$ execute correctly by assumption.
4. For all $v \in V_s$, if $s_e[v] \neq r_e[v]$, then v is invalid at s_e by Steps 1, 2 and 3 and by traversal invariant. □

4.3.6 Output message's validity

We finally show that, if faults are block-confined and some variable $v \in V_o$ is corrupt at s_e, then v is invalid with respect to $\ddot{v}$ at s_e.

Lemma 11 *Assume the traversal invariant holds at s_b and π does not crash. If there exists a variable $v \in V_o$ such that $s_e[v] \neq r_e[v]$, then $s_e[v] \neq s_e[\ddot{v}]$.*

Proof:
1. If v is determined by a fault, then $s_e[v] \neq s_e[\ddot{v}]$ by corruption coverage.
2. v is determined by an assignment by Step 1.
3. Case: Fault occurs in *Validate*
   3.1. *Initialization*, *Exec*$_1$, *Reset*, and *Exec*$_2$ blocks execute correctly by fault-frequency assumption.
   3.2. *Validate* block does not assign any variable $v \in V_o$ by definition.
   3.3. For all variable $v \in V_o$, $s_c[v] = r_e[v]$ by Steps 3.1 and 3.2.
   3.4. $s_e[v] = s_c[v] = r_e[v]$ by Steps 3.3 and 2.
4. Case: Fault occurs before *Validate*
   4.1. *Validate* block executes correctly by fault-frequency assumption.
   4.2. *Validate* block does not assign any variable $v \in V_o$ by definition.
   4.3. For all $v \in V_o$, v is valid at s_c, otherwise π crashes in the check of Line 71 by Step 4.1.
   4.4. For all $v \in V_o$, $s_e[v] = s_c[v]$ by Steps 4.2.
   4.5. For all $v \in V_o$, $s_e[v] = r_e[v]$.
      4.5.1. Assume there is a $v \in V_o$ such that $s_e[v] \neq r_e[v]$.
      4.5.2. v is invalid at s_e by Step 4.5.1 and Lemmas 3, 5, 8 and 9.
      4.5.3. A contradiction of Steps 4.5.2, 4.3 and 4.4.
5. For all $v \in V_o$, if $s_e[v] \neq r_e[v]$, then $s_e[v] \neq s_e[\ddot{v}]$ by Steps 1, 3 and 4. □

Lemma 12 *If the traversal invariant holds at state s_b and π does not crash, then the traversal invariant holds at state s_e.*

Proof: By Lemmas 3, 5, 8, 9, 10 and 11. □

4.4 Correctness with gates

We now show that the complete SEI-hardening guarantees error isolation. In contrast to the previous section, here we do not assume block-confined faults. In particular, in this section, faults either jump from within a block into another block or leave the traversal altogether by jumping out of the traversal.

Definition 48 (A fault jumps to another block) *A fault jumps to another block iff the fault occurs at some state s_f after execution of some instruction of a block B_1 and it corrupts the program counter pc to the value of some instruction in another block B_2.*

Definition 49 (A fault jumps out of the traversal) *A fault jumps out of the traversal iff the fault corrupts the program counter pc to the value of some instruction at or immediately after Line 84.*

Remember that Line 84 represents the exit point of the program, *i.e.*, messages may be externalized after this line.

The road map of our proof consists of the following steps:

1. We start by showing that if a fault occurs in a traversal A and the fault does not jump out of the traversal, then the fault is block-confined; hence, the results from block-confined faults apply.
2. Next, we show that even if a fault jumps out of the traversal A, local error exposure is preserved (although the traversal invariant might be violated).
3. Finally, we show that if a fault jumps out of traversal A, then any subsequent traversal B that modifies the state and does not crash π leads to a state in which the traversal invariant holds again.

4.4.1 Step 1: complete traversals

Let $Flags = \{cf_g, cf_p, cf_1, cf_r, cf_2, cf_c\}$ be the set of control-flow flags in SEI-hardening. To simplify the presentation, we incorporate the instructions of the gates as part of the blocks of Algorithm 1 such that:

- All instructions up to and including Line 8 belong to block *Filter*;
- $Prepare_1$ starts at the state s_g immediately after the execution of Line 8;
- $Prepare_2$ starts at the state s_p immediately after the execution of Line +2 of $Gate(cf_p, cf_g)$;
- $Exec_1$ starts at the state s_{x_1} immediately after the execution of Line +2 of $Gate(cf_1, cf_p)$;
- *Reset* starts at the state s_r immediately after the execution of Line +2 of $Gate(cf_r, cf_1)$;
- $Exec_2$ starts at the state s_{x_2} immediately after the execution of Line +2 of $Gate(cf_2, cf_r)$;
- *Validate* starts at the state s_c immediately after the execution of Line +2 of $Gate(cf_c, cf_2)$.

When the system starts, the first state of the first traversal is correct by Assumption 6. Therefore, we know that in the first traversal $s_b[cf_g] = \text{FALSE}$ and for every other flag $cf \in Flags$, $s_b[cf] = \text{TRUE}$ (Definition 45). We define the following predicate to express this condition at some state s.

$$\begin{aligned} Complete \triangleq\ & \land\, s[cf_g] = \text{FALSE} \lor \neg Valid(cf_g) \\ & \land\, \forall cf \in Flags \backslash \{cf_g\} : s[cf] = \text{TRUE} \lor \neg Valid(cf) \end{aligned}$$

Following directly from Algorithms 1 and 8, if *Complete* holds at s_b, then *Complete* holds at s_e as long as no fault occurs. As we show next, if *Complete* also holds at s_e when a fault occurs, then the fault cannot jump out of the traversal. Moreover, if a fault does not jump out of the traversal, then it can be equated as a block-confined fault.

Lemma 13 *Assume Complete holds at s_b, π does not crash, and a fault occurs before the first assignment of Line 5 and the fault jumps to another block. For all $v \in V_g$, $s_e[v] = s_b[v]$ or v is invalid at s_e.*

Proof:
1. Let s_f be the state immediately after the fault.
2. For all $cf \in Flags \backslash \{cf_g\}$, $s_f[cf] = \text{TRUE}$ or cf is invalid since *Complete* holds at s_b and by corruption coverage.
3. Only one fault occurs by fault-frequency assumption.
4. No instruction sets any $cf \in Flags \backslash \{cf_g\}$ to FALSE by Step 3 and since the fault jumps to another block by assumption and valid at s_a by assumption.
5. The fault jumps to some instruction after Line +0, otherwise π crashes by Steps 2 and 4.
6. No variable is modified by an instruction between s_b and s_e by Step 5.
7. For all $v \in V_g$, $s_e[v] = s_b[v]$ or v is invalid at s_e. □

By Lemma 13, we can disconsider faults occurring before the first assignment of Line 5 since the state stutters.

We now show that if the traversal invariant and, consequently, the *Complete* predicate, hold at s_b, and a fault does not jump out of the traversal, then the traversal invariant holds at s_e.

Lemma 14 *Assume Complete holds at s_b and π does not crash. If a fault occurs after the first assignment of Line 5 and the fault does not jump out of the traversal, then the first state of each block exists (Table 3) and they appear in the traversal in the order of Algorithm 1.*

Proof:
1. Let the s_a be the state immediately after the first assignment of Line 5.
2. $s_a[cf_c] =$ FALSE and valid at s_a by assumption.
3. The fault occurs after or at s_a by assumption.
4. Only one fault occurs by fault-frequency assumption.
5. State s_c exists and $s_c \prec s_e$.
   5.1. Line 82 is executed at state s since the fault does not jump out.
   5.2. cf_c is assigned to TRUE in $Gate(cf_c, cf_2)$, otherwise π crashes after s by Steps 2, 3 and 5.1.
   5.3. Let s_c be the state immediately after Line +2 in $Gate(cf_c, cf_2)$.
   5.4. $s_c \prec s_e$ since s_e is the last state of the traversal by definition.
6. State s_{x_2} exists and $s_{x_2} \prec s_c$.
   6.1. If no fault occurs before s_c, then cf_2 is assigned to TRUE in $Gate(cf_2, cf_r)$ at some state $s \prec s_c$.
   6.2. If no fault occurs after s_c, then $s_c[cf_2] =$ TRUE and valid at s_c, otherwise π crashes in the check after s_c (Line +3 in $Gate(cf_c, cf_2)$).
   6.3. If no fault occurs after s_c, cf_2 is assigned to TRUE in $Gate(cf_2, cf_r)$ at some state $s \prec s_c$ by Step 6.2.
   6.4. There exists a state $s_{x_2} = s$ such that $s_{x_2} \prec s_c$ by Steps 6.1 and 6.3.
7. State s_r exists and $s_r \prec s_{x_2}$.
   7.1. If no fault occurs before s_{x_2}, then cf_r is assigned to TRUE in $Gate(cf_r, cf_1)$, at some state $s \prec s_{x_2}$.
   7.2. If no fault occurs after s_{x_2}, then $s_{x_2}[cf_r] =$ TRUE and valid at s_{x_2}, otherwise π crashes in the check after s_{x_2} (Line +3 in $Gate(cf_2, cf_r)$).
   7.3. If no fault occurs after s_{x_2}, cf_r is assigned to TRUE in $Gate(cf_r, cf_1)$ at some state $s \prec s_{x_2}$ by Step 7.2.
   7.4. There exists a state $s_r = s$ such that $s_r \prec s_{x_2}$ by Steps 7.1 and 7.3.
8. State s_{x_1} exists and $s_{x_1} \prec s_r$.
   8.1. If no fault occurs before s_r, then cf_1 is assigned to TRUE in $Gate(cf_1, cf_p)$ at some state $s \prec s_r$.
   8.2. If no fault occurs after s_r, then $s_r[cf_1] =$ TRUE and valid at s_r, otherwise π crashes in the check after s_r (Line +3 in $Gate(cf_2, cf_r)$).
   8.3. If no fault occurs after s_r, cf_1 is assigned to TRUE in $Gate(cf_1, cf_p)$ at some state $s \prec s_r$ by Step 8.2.
   8.4. There exists a state $s_{x_1} = s$ such that $s_{x_1} \prec s_r$ by Steps 8.1 and 8.3.
9. State s_p exists and $s_p \prec s_{x_1}$.
   9.1. If no fault occurs before s_{x_1}, then cf_p is assigned to TRUE in *FirstGate*, at some state $s \prec s_{x_1}$.
   9.2. If no fault occurs after s_{x_1}, then $s_{x_1}[cf_p] =$ TRUE and valid at s_{x_1}, otherwise π crashes in the check after s_{x_1} (Line +3 in $Gate(cf_2, cf_r)$).
   9.3. If no fault occurs after s_{x_1}, cf_p is assigned to TRUE in *FirstGate* at some state $s \prec s_{x_1}$ by Step 9.2.
   9.4. There exists a state $s_p = s$ such that $s_p \prec s_{x_1}$ by Steps 9.1 and 9.3.
10. State s_g exists and $s_g \prec s_p$.
    10.1. If no fault occurs before s_p, then cf_g is assigned to TRUE in *FirstGate*, at some state $s \prec s_p$.
    10.2. If no fault occurs after s_p, then $s_p[cf_g] =$ TRUE and valid at s_p, otherwise π crashes in the check after s_p (Line +3 in $Gate(cf_p, cf_g)$).
    10.3. If no fault occurs after s_p, cf_g is assigned to TRUE in *FirstGate* at some state $s \prec s_p$ by Step 10.2.
    10.4. There exists a state $s_g = s$ such that $s_g \prec s_p$ by Steps 10.1 and 10.3.
11. State s_b exists and $s_b \prec s_g$ by definition of s_b (first state of a traversal).
12. s_b, s_g, s_p, s_{x_1}, s_r, s_{x_2}, s_c, and s_e exist and $s_b \prec s_g \prec s_p \prec s_{x_1} \prec s_r \prec s_{x_2} \prec s_c \prec s_e$. □

Lemma 15 *Assume Complete holds at s_b, π does not crash, and a fault occurs after the first assignment of Line 5 and the fault does not jump out of the traversal. Given the first state s of any block of Algorithm 1, no instruction of a preceding block is executed after s.*

Proof:
1. The first state of every block exists by Lemma 14.
2. No instruction preceding $Gate(cf_c, cf_2)$ is executed after s_c.
   2.1. Assume some instruction preceding $Gate(cf_c, cf_2)$ is executed at s after s_c.
   2.2. $s_c[cf_c] =$ TRUE and cf_c is valid at s_c by Step 1.

  2.3. π crashes at Line $+0$ of $Gate(cf_c, cf_2)$ after s either by Step 2.2 or by corruption coverage.
3. No instruction preceding $Gate(cf_2, cf_r)$ is executed after s_{x_2}.
  3.1. Assume some instruction preceding $Gate(cf_2, cf_r)$ is executed at s after s_{x_2}.
  3.2. $s_{x_2}[cf_2] =$ TRUE and cf_2 is valid at s_{x_2} by Step 1.
  3.3. π crashes at Line $+0$ of $Gate(cf_2, cf_r)$ after s either by Step 3.2 or by corruption coverage.
4. No instruction preceding $Gate(cf_r, cf_1)$ is executed after s_r.
  4.1. Assume some instruction preceding $Gate(cf_r, cf_1)$ is executed at s after s_r.
  4.2. $s_r[cf_r] =$ TRUE and cf_r is valid at s_r by Step 1.
  4.3. π crashes at Line $+0$ of $Gate(cf_r, cf_1)$ after s either by Step 4.2 or by corruption coverage.
5. No instruction preceding $Gate(cf_1, cf_p)$ is executed after s_{x_1}.
  5.1. Assume some instruction preceding $Gate(cf_1, cf_p)$ is executed at s after s_{x_1}.
  5.2. $s_{x_1}[cf_1] =$ TRUE and cf_1 is valid at s_{x_1} by Step 1.
  5.3. π crashes at Line $+0$ of $Gate(cf_2, cf_r)$ after s either by Step 5.2 or by corruption coverage.
6. No instruction preceding $Gate(cf_p, cf_g)$ is executed after s_p.
  6.1. Assume some instruction preceding $Gate(cf_p, cf_g)$ is executed at s after s_p.
  6.2. $s_p[cf_p] =$ TRUE and cf_p is valid at s_p by Step 1.
  6.3. π crashes at Line $+0$ of $Gate(cf_p, cf_g)$ after s either by Step 6.2 or by corruption coverage.
7. No instruction preceding *FirstGate* is executed after s_g.
  7.1. Assume some instruction preceding Line 8 (*FirstGate*) is executed at s after s_g.
  7.2. $s_g[cf_g] =$ TRUE and cf_g is valid at s_g by Step 1.
  7.3. π crashes at Line 6 after s either by Step 7.2 or by corruption coverage.
8. No instruction of a preceding block is executed after the first instruction of the following block by Steps 2, 3, 4, 5, 6 and 7. □

Lemma 15 *does not* disallow blocks to be executed twice as long as the fault does not cross the border of some gate. For example, $Exec_1$ could be executed again after $Gate(cf_1, cf_p)$ and before $Gate(cf_r, cf_1)$. In fact, if a fault does not jump out of a traversal, then the fault is block-confined (as in Assumption 7). Therefore, the results of Section 4.3 hold here as well.

Lemma 16 *If the traversal invariant holds at s_b, a fault occurs and does not jump out of the traversal, and π does not crash, then the traversal invariant holds at s_e.*

Proof:
1. *Complete* holds at s_b by the traversal invariant.
2. The first state of every block exists and are the order of Algorithm 1 by Step 1 and by Lemma 14.
3. Faults are block-confined by Step 1 and by Lemma 15.
4. The traversal invariant holds at s_e by Steps 2 and 3 and Lemma 12. □

4.4.2 Step 2: partial traversals

There are four cases to consider when a fault jumps out of a traversal:

Skipped traversal: A fault occurs at some state before the execution of Line 8, *i.e.*, before s_g: only variables in $Flags \backslash \{cf_g\}$ might be assigned before such fault occurs, and no variable is assigned after the fault occurs. If cf_g is valid at s_e, then $s_e[cf_g] =$ FALSE.

Initialized traversal: A fault occurs after s_g and before the execution of Line $+2$ of $Gate(cf_p, cf_g)$, *i.e.*, before s_p: The bookkeeping data structures and the control-flow flags of SEI-hardening are reset, but no variable of the actual program state are modified.

Partial traversal: A fault occurs after s_p and before the execution of Line $+2$ of $Gate(cf_c, cf_2)$, *i.e.*, before s_c: Some of the blocks modifying state (*i.e.*, $Exec_1$, *Reset* and $Exec_2$) is executed partially. The fault jumps out before entering into the *Validate* block.

Terminal traversal: A fault occurs after s_c: All blocks are executed correctly except of *Validate*.

We now show that for each of these cases local error exposure (Property 3) is preserved.

Lemma 17 *Assume the traversal invariant holds at s_b, π does not crash, and a fault occurs before the execution of Line 8. If the fault jumps out of the traversal, then for all $v \in V_s$, $s_e[v] = s_b[v]$ or v is invalid at s_e.*

Proof:
1. Let s be the state immediately after the fault occurs.
2. The fault after or during the execution of Line 5.
   2.1. Otherwise the traversal simply stutters by Lemma 13 and can be ignored.
3. For some $cf \in \mathit{Flags}\backslash\{cf_g\} : s[cf] = \text{FALSE}$ or cf is invalid at s by corruption coverage and since some instructions of Line 5 are executed by Step 2.
4. The fault jumps to the last line of p_h, otherwise π crashes at Line 82 by Step 3.
5. No variable $v \in V_s$ is modified by an instruction between s_b and s_e by Step 4 and by definition of p_h.
6. For all $v \in V_s$, $s_e[v] = s_b[v]$ or v is invalid at s_e by Step 5 and by corruption coverage. □

Note that by Lemma 17, no variable in V_s is modified, but some variable in $\mathit{Flags}\backslash\{cf_g\}$ might be reset. Since no variable in V_s local error exposure is not violated since it depends only on V_o and $V_o \subset V_s$.

Lemma 18 *Assume the traversal invariant holds at s_b, π does not crash, and a fault occurs after s_g and before the execution of Line +2 of $Gate(cf_p, cf_g)$. If the fault jumps out of the traversal, then for all $v \in V_s$, $s_e[v] = r_b[v]$ or v is invalid at s_e.*

Proof:
1. The fault occurs at state s before Line +2 in $Gate(cf_p, cf_g)$ and after *FirstGate*, otherwise *Initialized* does not hold at s_e.
2. Only *Filter*, *FirstGate* and $\mathit{Prepare}_1$ are executed by Step 1.
3. For all $v \in V_i$, $s[v] = r_b[v]$, $s[v] = s[\bar{v}]$, $s[v] = s[\ddot{v}]$ by Step 2, traversal invariant, and definition of *Filter*.
4. For all $v \in V_s$, $s[v] = r_b[v]$ or v is invalid at s by Steps 2 and 3 and by traversal invariant.
5. For all $v \in V_s$, $s_e[v] = r_b[v]$ by Step 4 or v is invalid at s_e by corruption coverage. □

Lemma 19 *Assume the traversal invariant holds at s_b, π does not crash, and a fault occurs after s_c. If the fault jumps out of the traversal, then for all $v \in V_s$, $s_e[v] = r_e[v]$ or v is invalid at s_e.*

Proof:
1. The fault occurs at state s after Line +2 in $Gate(cf_c, cf_2)$.
2. All blocks except of *Validate* execute correctly by Step 1.
3. For all variable $v \in V_s$, $s[v] = r_e[v]$ by Step 2 and traversal invariant.
4. For all $v \in V_s$, $s_e[v] = r_e[v]$ or v is invalid at s_e by Step 3 or by corruption coverage. □

Lemma 20 *Assume the traversal invariant holds at s_b, π does not crash, and a fault occurs after s_p and before the execution of Line +2 of $Gate(cf_c, cf_2)$. If the fault jumps out of the traversal and $\exists v \in V_o$: $s_e[v] \neq r_e[v]$, then $s_e[v] \neq s_e[\ddot{v}]$.*

Proof:
1. Assume by contradiction that some $v \in V_o$ is such that $s_e[v] \neq r_e[v]$ and $s_e[v] = s_e[\ddot{v}]$.
2. If the fault corrupts v, then $s_e[v] \neq s_e[\ddot{v}]$ by corruption coverage and since the fault jumps out of the traversal.
3. v is assigned at some state s before the fault by Steps 1 and 2.
4. The fault does not modify v after s by Step 2.
5. No fault occurs before s' by fault-frequency assumption and by Step 3.
6. Case: Fault occurs in Exec_1.
   6.1. $s[\ddot{v}] \neq s[v]$ by definition of assignment to $v \in V_o$ (Rule 5).
   6.2. $s = s_e$ since fault jumps out and by Step 4.
   6.3. A contradiction by Steps 1, 6.1 and 6.2.
7. Case: Fault occurs in *Reset*.
   7.1. Exec_1 executes correctly by case assumption.
   7.2. If v is not assigned in *Reset*, $s_e[\ddot{v}] \neq s_e[v]$ by definition of assignment to $v \in V_o$ and by Step 7.1; a contradiction.
   7.3. v is assigned in *Reset* at state s by Steps 3, 7.1 and 7.2.
   7.4. $s'[v] = s_b[v]$ by definition of *Reset* and Step 7.3.
   7.5. $s'[\ddot{v}] \neq s_b[v]$ by definition of assignment to $v \in V_o$.
   7.6. $s'[\ddot{v}] \neq s'[v]$ by Steps 7.4 and 7.5.
   7.7. $s' = s_e$ since fault jumps out and by Step 4.

7.8. A contradiction by Steps 7.1, 7.2, 7.6 and 7.7.

8. Case: Fault occurs in $Exec_2$.

8.1. $Exec_1$ and *Reset* execute correctly by case assumption.

8.2. If v is not assigned in $Exec_2$, $s_e[\ddot{v}] \neq s_e[v]$ by definition of assignment to $v \in V_o$ in $Exec_1$ and by Step 8.1; a contradiction.

8.3. v is assigned in $Exec_2$ at state s by Steps 3, 8.1 and 8.2.

8.4. $s'[v] = r_e[v]$ by Steps 8.3.

8.5. $s' = s_e$ since fault jumps out and by Step 4.

8.6. A contradiction by Steps 8.2, 8.4 and 8.5.

9. If $\exists v \in V_o : s_e[v] \neq r_e[v]$, then $s_e[v] \neq s_e[\ddot{v}]$ by Steps 6, 7 and 8. □

Lemmas 17, 18, 19 and 20 show that if a fault jumps out of the traversal the variables in V_o do not represent a valid output message, *i.e.*, local error exposure (Property 3) holds at state s_e. Nevertheless, the state s_e is potentially corrupt and the traversal invariant might not hold at s_e.

4.4.3 Step 3: the limited effect of multiple faults

We now turn to the cases in which a fault jumps out of a traversal A, and a subsequent traversal B executes. We have seen that local error exposure holds at the final state of traversal A. Our next goal is to show that if π does not crash during a subsequent traversal B, then the traversal invariant holds at the final state of traversal B. We use the following notation to differentiate the state of both traversals: Non-primed states (*e.g.*, s_b, r_b, s_g, s_p, s_{x_1}) are states in traversal A; primed states (*e.g.*, s_b', r_b', s_g', s_p', s_{x_1}') are states in traversal B.

The key insight of this section is that multiple corruption faults have a limited effect. By Assumption 3, at most one fault occurs per traversal, but between traversals several faults might occur. We will show that a sequence of faults formed by one fault that jumps out of traversal A followed by any number of faults between traversals and, finally, followed by zero or one fault in traversal B behaves *similarly to a single block-confined fault*. Nevertheless, such fault sequences are *not equivalent* to single block-confined faults. They can render input messages to be omitted and output messages to be duplicated. Such failures are, however, in the crash-stop fault model and assumed to be tolerated by any algorithm being hardened (see Section 2.3).

We start with the case of a fault jumping out of traversal A *before* the execution of Line 8. The control-flow flags might be partially initialized. In particular, cf_g is still FALSE at state s_e[3] since it is only set to TRUE at Line 8 and the fault *jumps out* of the traversal. If no fault occurs in traversal B, partially initialized flags are fully initialized and the traversal executes normally. Note that the input message of traversal A is not processed. Therefore, the reference initial state r_b' of the subsequent traversal B is the same reference initial state r_b of traversal A except that that the input message in V_i is different since messages are received between traversals. The next lemma asserts that if a fault does occur in traversal B, then the traversal invariant holds at s_e'.

Lemma 21 *Assume traversal invariant holds at state s_b of traversal A, a fault occurs at some state s_f before the execution of Line 8 and jumps out of traversal A, and π does not crash. If a fault occurs in traversal B, then for all $v \in V_g$, $s'_e[v] = r'_e[v]$ or v is invalid at s'_e.*

Proof:

1. Let s'_f be the state immediately after a fault occurs in traversal B.
2. $\forall v \in V_g : s_e[v] = s_f[v]$ or v is invalid at s_e by definition of s_f and by corruption coverage.
3. No instruction modifies any $v \in V_g \backslash V_i$ until s'_f by definition.
4. $\forall v \in V_g \backslash V_i : s'_f[v] = s_f[v]$ or v is invalid at s'_f by Step 2 and 3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
5. Case: The fault occurs in traversal B before the execution of Line 8.

5.1. If s'_f is in *Filter* block up to Line 8, then the sequence of faults from traversal A to traversal B are equivalent to a single block-confined fault by Step 4; the results follow from Lemma 12.

5.2. If s'_f is after Line 8, then the state s'_g of traversal B exists by Lemma 14; a contradiction with the fact the state at which the fault occurs precedes Line 8 by case assumption and s'_f succeeds Line 8 by step assumption.

6. Case: A fault occurs in traversal B after the execution of Line 8.

6.1. *Filter* and *FirstGate* execute correctly in traversal B by case assumption since Line 8 is the last line of *FirstGate*.

[3]The state immediately after a fault that jumps out of a traversal is the final state of that traversal.

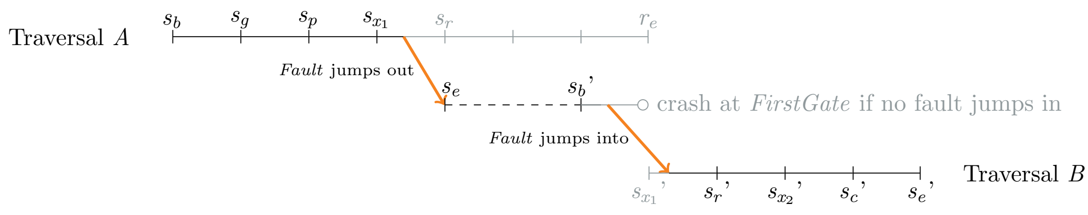


Figure 9: Jump-out jump-in fault sequence: If a fault jumps out of traversal A, then in the subsequent traversal B: (1) another fault has to occur; and (2) this fault has to jump into the same block in which traversal A was left (see Lemma 22).

6.2. For all $v \in V_g \setminus V_i : s'_g[v] = s_g[v]$ or v is invalid by Step 4 and by case assumption.
6.3. Let r_g be the reference state after the execution of Line 8 of traversal A.
6.4. For all $v \in V_g \setminus V_i : s'_g[v] = r_g[v]$ or v is invalid by Step 6.2.
6.5. s_g' is the same state as s_g, but with a potentially different input message in V_i by Step 6.4 and since between two traversal a new message might be received.
6.6. For all $v \in V_g$, $s'_e[v] = r'_e[v]$ or v is invalid at s'_e by Step 6.5.
7. For all $v \in V_g$, $s'_e[v] = r'_e[v]$ or v is invalid at s'_e by Steps 5 and 6. □

We now show that also for the case of a fault jumping out of traversal A *after* Line 8, the traversal invariant holds at s_e'. The control-flow flag cf_g is TRUE at once the fault occurs in traversal A since it is set at Line 8. It is easy to see that if no fault occurs in the subsequent traversal B, π crashes at the checks of *FirstGate* because either $s'_b[cf_g] =$ TRUE or cf_g is invalid at s_b'. In fact, if any instruction of blocks $Exec_1$, *Reset* or $Exec_2$ are to be executed in traversal B, a fault has to occur before Line 3 completely jumping over the checks of the *FirstGate*. For that we define a fault that "jumps in" a traversal as follows.

Definition 50 (A fault jumps in) *A fault jumps in the traversal iff a fault occurs at some line before Line 3 corrupting the program counter pc to the value of some instruction after Line 6, but before Line 84.*

The next lemma (Lemma 22) asserts that not only that fault has to occur in traversal B, but also that the fault has to *jump into exactly the same block* at which traversal A was left. Figure 9 exemplifies a fault occurring in $Exec_1$ jumping out of traversal A. A fault has to occur jump into traversal B, otherwise π crashes in the checks of *FirstGate* since $s'_b[cf_g] =$ TRUE or invalid at s_b'. Following the example of the figure, if the fault jumps into B before $Gate(cf_1, cf_p)$, *i.e.*, before s_{x_1}', then π crashes at Line +0 of $Gate(cf_1, cf_p)$ because $s'_b[cf_1] =$ TRUE or invalid at s_b' since the fault in traversal A jumped out after $Gate(cf_1, cf_p)$, *i.e.*, s_{x_1}. If the fault jumps into B after $Gate(cf_r, cf_1)$, *i.e.*, after s_r', then π crashes at Line +3 of $Gate(cf_2, cf_r)$ because $s'_b[cf_r] =$ FALSE or invalid at s_b' since the fault in traversal A jumped out before $Gate(cf_r, cf_1)$, *i.e.*, before s_r. Hence, the only block into which the fault can jump is $Exec_1$, *i.e.*, after s_{x_1}' and before s_r'.

The consequence of Lemma 22 is important. The faults of two consecutive traversals behave similarly to a single block-confined fault. Moreover, since multiple faults might occur between traversals, the sequence of faults formed by a fault jumping out of traversal A followed by zero or more faults between traversals A and B and, finally, followed by a fault jumping into traversal B behave as a single block-confined fault.

Lemma 22 *Assume traversal invariant holds at state s_b of traversal A, a fault occurs at some state s_f after the execution of Line 8 and jumps out of traversal A, and π does not crash. If a fault jumps into traversal B at state s'_f, then s'_f is in the same block as s_f.*

Proof:
1. $s'_e[cf_g] =$ FALSE or cf_g is valid at s_e'.
1.1. $s_f[cf_g] =$ TRUE or cf_g is valid at s_f since the fault in traversal A occurs after Line 8.
1.2. $s_e[cf_g] =$ TRUE or cf_g is invalid at s_e by Step 1.1 and since the fault jumps out of traversal A and by corruption coverage.
1.3. $s'_b[cf_g] =$ TRUE or cf_g is invalid at s_b' by Step 1.2 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
1.4. A fault has to occur in traversal B at some state s before the execution of Line 3 and jump into traversal B after Line 6, otherwise π crashes by Step 1.3.

1.5. $s_f'[cf_g] = \text{TRUE}$ or cf_g is invalid at s_f' by Step 1.3 and 1.4 and by corruption coverage.
1.6. If s_f' is some state after Line 73, then no variable in V_g is modified in B; we can ignore such cases since B is a series of stuttering steps.
1.7. If s_f' is some state before Line 73, then $s_e'[cf_g] = \text{FALSE}$ and cf_g is valid at s_e' by definition of Line 73 and by fault-frequency assumption.

2. CASE: s_f in *Validate* before Line 73.
2.1. $\forall cf \in \mathit{Flags} : s_f[cf] = \text{TRUE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault-frequency assumption, and every $cf \in \mathit{Flags}$ is assigned TRUEby an instruction before Line 73 by Algorithm 8.
2.2. No instruction modifies any $cf \in \mathit{Flags}$ from s_f until s_f' by Definitions 49 and 50.
2.3. $\forall cf \in \mathit{Flags} : s_f'[cf] = \text{TRUE}$ or cf is invalid at s_f' by Step 2.1 and 2.2 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
2.4. s_f' must be after Line +0 of $\mathit{Gate}(cf_c, cf_2)$, otherwise π crashes by Step 2.3.
2.5. s_f' must be before Line 73, otherwise $s_e'[cf_g] = \text{TRUE}$ or cf_g is invalid at s_e' by Step 2.3; a contradiction with Step 1.
2.6. s_f' is in *Validate* by Steps 2.4 and 2.5.

3. CASE: s_f in Exec_2.
3.1. $\forall cf \in \mathit{Flags}\backslash\{cf_c\} : s_f[cf] = \text{TRUE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault-frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before Exec_2.
3.2. $s_f[cf_c] = \text{FALSE}$ and cf_c is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before Exec_2.
3.3. No instruction modifies any $cf \in \mathit{Flags}$ from s_f until s_f' by Definitions 49 and 50.
3.4. $\forall cf \in \mathit{Flags}\backslash\{cf_c\} : s_f'[cf] = \text{TRUE}$ or cf is invalid at s_f' by Step 3.1 and 3.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
3.5. $s_f'[cf_c] = \text{FALSE}$ or cf_c is invalid at s_f' by Step 3.2 and 3.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
3.6. s_f' must be after Line +0 of $\mathit{Gate}(cf_2, cf_r)$, otherwise π crashes by Step 3.4.
3.7. s_f' must be before Line 73, otherwise $s_e'[cf_g] = \text{TRUE}$ or cf_g is invalid at s_e' by Step 3.3; a contradiction with Step 1.
3.8. s_f' must be before or at Line +2 of $\mathit{Gate}(cf_c, cf_2)$, otherwise π crashes after Line 73 by Steps 3.5 and 3.7.
3.9. s_f' is in Exec_2 by Step 3.8.

4. CASE: s_f in *Reset*.
4.1. $\forall cf \in \mathit{Flags}\backslash\{cf_2, cf_c\} : s_f[cf] = \text{TRUE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before *Reset*.
4.2. $\forall cf \in \{cf_2, cf_c\} : s_f[cf] = \text{FALSE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before *Reset*.
4.3. No instruction modifies any $cf \in \mathit{Flags}$ from s_f until s_f' by Definitions 49 and 50.
4.4. $\forall cf \in \mathit{Flags}\backslash\{cf_2, cf_c\} : s_f'[cf] = \text{TRUE}$ or cf is invalid at s_f' by Step 4.1 and 4.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
4.5. $\forall cf \in \{cf_2, cf_c\} : s_f'[cf] = \text{FALSE}$ or cf is invalid at s_f' by Step 4.2 and 4.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
4.6. s_f' must be after Line +0 of $\mathit{Gate}(cf_r, cf_1)$, otherwise π crashes by Step 4.4.
4.7. s_f' must be before Line 73, otherwise $s_e'[cf_g] = \text{TRUE}$ or cf_g is invalid at s_e' by Step 4.3; a contradiction with Step 1.
4.8. s_f' must be before or at Line +2 of $\mathit{Gate}(cf_2, cf_r)$, otherwise π crashes after Line 73 by Steps 4.5 and 4.7.
4.9. s_f' is in *Reset* by Step 4.8.

5. CASE: s_f in Exec_1.
5.1. $\forall cf \in \mathit{Flags}\backslash\{cf_r, cf_2, cf_c\} : s_f[cf] = \text{TRUE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before Exec_1.
5.2. $\forall cf \in \{cf_r, cf_2, cf_c\} : s_f[cf] = \text{FALSE}$ and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in \mathit{Flags}$ is assigned by an instruction before Exec_1.
5.3. No instruction modifies any $cf \in \mathit{Flags}$ from s_f until s_f' by Definitions 49 and 50.
5.4. $\forall cf \in \mathit{Flags}\backslash\{cf_r, cf_2, cf_c\} : s_f'[cf] = \text{TRUE}$ or cf is invalid at s_f' by Step 5.1 and 5.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
5.5. $\forall cf \in \{cf_r, cf_2, cf_c\} : s_f'[cf] = \text{FALSE}$ or cf is invalid at s_f' by Step 5.2 and 5.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.
5.6. s_f' must be after Line +0 of $\mathit{Gate}(cf_1, cf_p)$, otherwise π crashes by Step 5.4.
5.7. s_f' must be before Line 73, otherwise $s_e'[cf_g] = \text{TRUE}$ or cf_g is invalid at s_e' by Step 5.3; a

contradiction with Step 1.

5.8. s'_f must be before or at Line +2 of $Gate(cf_r, cf_1)$, otherwise π crashes after Line 73 by Steps 5.5 and 5.7.

5.9. s'_f is in $Exec_1$ by Step 5.8.

6. CASE: s_f in $Prepare_2$.

6.1. $\forall cf \in Flags \backslash \{cf_1, cf_r, cf_2, cf_c\} : s_f[cf] = $ TRUE and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in Flags$ is assigned by an instruction before $Prepare_2$.

6.2. $\forall cf \in \{cf_1, cf_r, cf_2, cf_c\} : s_f[cf] = $ FALSE and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in Flags$ is assigned by an instruction before $Prepare_2$.

6.3. No instruction modifies any $cf \in Flags$ from s_f until s'_f by Definitions 49 and 50.

6.4. $\forall cf \in Flags \backslash \{cf_1, cf_r, cf_2, cf_c\} : s'_f[cf] = $ TRUE or cf is invalid at s'_f by Step 6.1 and 6.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.

6.5. $\forall cf \in \{cf_1, cf_r, cf_2, cf_c\} : s'_f[cf] = $ FALSE or cf is invalid at s'_f by Step 6.2 and 6.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.

6.6. s'_f must be after Line +0 of $Gate(cf_p, cf_g)$, otherwise π crashes by Step 6.4.

6.7. s'_f must be before Line 73, otherwise $s'_e[cf_g] = $ TRUE or cf_g is invalid at s_e' by Step 6.3; a contradiction with Step 1.

6.8. s'_f must be before or at Line +2 of $Gate(cf_1, cf_p)$, otherwise π crashes after Line 73 by Steps 6.5 and 6.7.

6.9. s'_f is in $Prepare_2$ by Step 6.8.

7. CASE: s_f in $Prepare_1$.

7.1. $\forall cf \in Flags \backslash \{cf_p, cf_1, cf_r, cf_2, cf_c\} : s_f[cf] = $ TRUE and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in Flags$ is assigned by an instruction before $Prepare_1$.

7.2. $\forall cf \in \{cf_p, cf_1, cf_r, cf_2, cf_c\} : s_f[cf] = $ FALSE and cf is valid at s_f since no fault occurs before s_f in A by fault frequency assumption, and every $cf \in Flags$ is assigned by an instruction before $Prepare_1$.

7.3. No instruction modifies any $cf \in Flags$ from s_f until s'_f by Definitions 49 and 50.

7.4. $\forall cf \in Flags \backslash \{cf_p, cf_1, cf_r, cf_2, cf_c\} : s'_f[cf] = $ TRUE or cf is invalid at s'_f by Step 7.1 and 7.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.

7.5. $\forall cf \in \{cf_p, cf_1, cf_r, cf_2, cf_c\} : s'_f[cf] = $ FALSE or cf is invalid at s'_f by Step 7.2 and 7.3 and by corruption coverage, irrespective of how many faults occurred between traversals A and B.

7.6. s'_f must be after Line 6, otherwise π crashes by Step 7.4.

7.7. s'_f must be before Line 73, otherwise $s'_e[cf_g] = $ TRUE or cf_g is invalid at s_e' by Step 7.3; a contradiction with Step 1.

7.8. s'_f must be before or at Line +2 of $Gate(cf_p, cf_g)$, otherwise π crashes after Line 73 by Steps 7.5 and 7.7.

7.9. s'_f is in $Prepare_1$ by Step 7.8.

□

Note that the case of fault in traversal A occurring after Line 73 can be ignored because in such case all blocks execute correctly. Also note that if s_f in *Filter*, then the fault occurs before Line 8, and the results follow from Lemma 21.

Lemma 23 *Assume the traversal invariant holds at state s_b of traversal A, a fault occurs at some state s_f after Line 8 and jumps out of traversal A, and π does not crash. If a fault jumps into traversal B, then the for all $v \in V_g \backslash V_i$, $s'_f[v] = s_f[v]$ or v is invalid at s'_f.*

Proof:

1. No variable in $V_g \backslash V_i$ is modified by assignment between s_f and s_e of traversal A by Definition 49.
2. No variable in $V_g \backslash V_i$ is modified by assignment between s'_b and s'_f of traversal B by Definition 50.
3. No variable in $V_g \backslash V_i$ is modified by assignment between s_e and s_b' since variables in $V_g \backslash V_i$ are only modified inside traversals.
4. For all $v \in V_g \backslash V_i$, $s'_f[v] = s_f[v]$ or v is invalid at s'_f by Steps 1, 2 and 3 and by corruption coverage. □

Finally, we show that if the traversal invariant holds at state s_b of a traversal A, a fault jumps out of traversal A, and another fault jumps into traversal B, then the traversal invariant holds at s_e'. In particular, it holds that $s'_e[v] = r_e[v]$ for all variable $v \in V_g$ that is valid at s_e'; or $s'_e[v] = r'_e[v]$ for all variable $v \in V_g$ that is valid at s_e'.

Lemma 24 *Assume the traversal invariant holds at state s_b of traversal A, a fault occurs at some state s_f after Line 8 and jumps out of traversal A. If a fault jumps into traversal B at state s'_f and π does not crash, then $\forall v \in V_g \backslash V_i : s'_e[v] = r_e[v] \vee s'_e[v] = r'_e[v]$ or v is invalid at s_e'.*

Proof:
1. No fault occurs in traversal A before s_f by fault-frequency assumption.
2. No fault occurs in traversal B after s'_f by fault-frequency assumption.
3. For all $v \in V_g \backslash V_i$, $s'_f[v] = s_f[v]$ or v is invalid at s'_f by Lemma 23.
4. N, O and U are valid at s'_e since no fault occurs after s'_f and s'_f occurs before *LastGate*.
5. Case: s_f in *Validate* before Line 73.
   5.1. State s'_f is in block *Validate* by Lemma 22.
   5.2. No assignment to $V_g \backslash V_i$ occurs after s_c until s'_f by definition of *Validate* and by Step 3.
   5.3. No assignment to V_g occurs after s'_f by definition of *Validate*.
   5.4. $\forall v \in V_g \backslash V_i : s_e[v] = s_f[v] = r_e[v]$ by Step 1 and by case assumption.
   5.5. $\forall v \in V_g \backslash V_i : s'_e[v] = r_e[v]$ by Steps 5.2, 5.3 and 5.4.
6. Case: s_f in *Exec*$_2$.
   6.1. State s'_f is in block *Exec*$_2$ by Lemma 22.
   6.2. Case: $v \in N$ and $v \in U$ at s'_f.
      6.2.1. $s'_e[v] = s'_e[N(v)]$, otherwise π crashes in *Validate*.
      6.2.2. $s'_e[N(v)] = s'_f[N(v)] = s_f[N(v)]$ by Step 3.
      6.2.3. $s_f[N(v)] = r_e[v]$ by Step 1 and 4.
      6.2.4. $s'_e[v] = r_e[v]$ by Steps 6.2.1, 6.2.2, and 6.2.3.
   6.3. Case: $v \in N$ and $v \notin U$ at s'_f; or $v \notin N$ and $v \in U$ at s'_f.
      6.3.1. π crashes at *Validate*; a contradiction
   6.4. Case: $v \notin N$ and $v \notin U$ at s'_f.
      6.4.1. v is assigned in *Exec*$_2$ at some state s', otherwise $s'_e[v] = s_e[v] = s_b[v]$.
      6.4.2. $s'_f \preceq s'$ by Definition 50 and fault-frequency assumption.
      6.4.3. No fault occurs after s' by Step 6.4.2.
      6.4.4. π crashes at the check of Line +2 of Rule 4; a contradiction.
7. Case: s_f in *Reset*.
   7.1. State s'_f is in block *Reset* by Lemma 22.
   7.2. *Reset* does not read or write any $v \in V_i$ by definition of *Reset*.
   7.3. $\forall v \in V_g : s'_e[v] = r_e[v]$ by Lemmas 22 and 8 and Steps 7.1 and 7.2.
8. Case: s_f in *Exec*$_1$.
   8.1. State s'_f is in block *Exec*$_1$ by Lemma 22.
   8.2. $\forall v \in V_g : s'_e[v] = r'_e[v]$ by Lemmas 22 and 5 and Steps 8.1 and 3.
9. Case: s_f in *Prepare*$_2$ or s_f in *Prepare*$_1$.
   9.1. $\forall v \in V_g : s'_e[v] = r'_e[v]$ by Lemmas 22 and 3 and Step 3.

□

We now conclude our proof that SEI-hardening guarantees error isolation.

Theorem 2 *SEI-hardening guarantees error isolation under the Assumptions 2, 3, 4, 5, and 6.*

Proof:
1. Local error exposure (Property 3)
   If a fault jumps out of the traversal the result holds by Lemmas 1, 17, 21 and 24. If a fault does not jump out of the traversal, the result holds by Lemmas 1 and 16.
2. Local error filtering (Property 4)
   If an invalid message is received a correct process discards the message in *Filter* before any variable $v \in V_g$ is modified by definition of Algorithms 1 and 2.
3. Accuracy (Property 2)
   A message is only discarded by a correct process if it is invalid by definition of Algorithms 1 and 2. Process π only aborts if an error is detected by Rules 1-6 and Algorithms 1, 2, 5 and 8.
4. Under Assumptions 2, 3, 4, 5, and 6, SEI-hardening guarantees error isolation by Steps 1, 2 and 3 and Theorem 1. □

5 Multithreaded SEI-hardening

In this section, we introduce multithreading support into SEI. Our approach uses two-phase locking [33] to isolate the state used by a traversal from other concurrent traversals. In particular, locks are acquired and held until both executions and the final checks of a traversal finish. Additionally, since threads can read corrupt values from faulty threads that skip lock acquisition, we add a barrier after the *Validate* block, which enforces that a traversal only terminates after all other concurrently executed traversals have finished their *Validate* blocks.

We start by discussing further model refinements necessary to support multithreading; next, we prove the approach correct; and, finally, we discuss the limitations of our multithreading support and a possible alternative solution.

5.1 Model refinements

Definition 51 (Multithreaded process) *A process π consists of multiple treads $\{\tau_0, \ldots, \tau_t\}$. Each thread τ_i executes the Next state transitions on the variable set $V_i \subset V$. Threads share a subset V_{shr} of their variables. No input or output variable is shared among threads.*

Assumption 8 (Mutual exclusion and lock hierarchies) *Threads access variables in V_{shr} only in critical sections. Mutual exclusion is obtained using locks and locks are acquired in a consistent order,* e.g.*, via lock hierarchies [17].*

Given the way SEI handles locks, the behavior of multithreaded processes is equivalent to a single-threaded process. Therefore, we can adopt the notion of correct message for single-threaded processes (Definition 3). However, for completeness, we extend the definition of correct message to a multithreaded process.

Definition 52 (Refinement of a generation history of a message) *Let π be a multithreaded process, m be a message sent by π, and h the generation history for m. A refinement $\widehat{h}$ of h is the sequence of local steps π executes in a run where π is correct and receives each message in h until m is sent.*

Definition 53 (Correct multithreaded generation history of a message) *Let π be a faulty multithreaded process. Let m be a message sent by π.*

- *If π has sent no message m' before m such that m' has a correct generation history, then all generation histories of m are correct for m.*
- *Else, for each output message m' preceding m, let H be set of correct generation histories of m'. A generation history of m is correct if and only if it extends some generation history in H.*

In addition, given two output messages m_1 and m_2 having two sets of correct generations histories H_{m_1} and H_{m_2} respectively, it must hold that for each refinement $\widehat{h_1}$ of $h_1 \in H_{m_1}$ (resp. $\widehat{h_2}$ of $h_2 \in H_{m_2}$) there must exist a refinement $\widehat{h_2}$ of $h_2 \in H_{m_2}$ (resp. $\widehat{h_1}$ of $h_1 \in S_{m_1}$), such that either $\widehat{h_1}$ extends $\widehat{h_2}$ or $\widehat{h_2}$ extends $\widehat{h_1}$.

The first two conditions are the same as Definition 3. The third is specific to multithreaded processes.

In presence of multiple threads, we can have multiple concurrent traversals, at most one per thread. The *fault frequency* assumption with multiple traversals becomes the following.

Assumption 9 (Fault frequency with multiple threads) *Given a set of concurrent traversals $E = \{E_1, \ldots, E_n\}$ executed by different threads, at most one fault occurs between the earliest beginning and the latest end of a traversal in E.*

The state of a lock l indicates the thread t currently holding l. We consider three lock primitives for a lock l: *Acquire*(l), *Release*(l), *Holding*(l). The first primitive acquires l if it is available. It returns a boolean indicating success. Locks can be acquired multiple times. The second primitive releases the lock. It returns false if it is invoked by a thread that is not currently holding l and true otherwise. The last primitive returns a boolean indicating whether the invoking thread holds l.

A *critical section* C is a sequence of instructions that must be executed in mutual exclusion. In a correct execution, C is only executed by threads holding a set of locks L_C.

```
    add at the end of the Initialization blocks
+1      if $\neg Check(i) \vee \neg Check(c[i])$ then Abort
+2      $c[i] \leftarrow c[i] + 1$
    after executing Validate
+1      foreach $l \in Q$ do
+2          $Release(l)$
+3      $c[i] \leftarrow c[i] + 1$
+4      $c_{snap} \leftarrow c$
+5      if $\neg Check(c_{snap})$ then Abort
+6      foreach $j \neq i$ such that $c_{snap}[j]$ is even do
+7          wait until $c[j] > c_{snap}[j] \vee \neg Check(c[j])$
```

Algorithm 9: Hardening rules for thread τ_i

```
    before $Acquire(l)$ in $Exec_1$
+1      add $l$ to set $Q$
    around $Acquire(l)$ in $Exec_2$
+1      do nothing
    around $Release(l)$ during $Exec_1$ or $Exec_2$
+1      if $\neg Check(l) \vee \neg Holding(l)$ then Abort
```

Algorithm 10: Intercepted operations

5.2 Algorithm extensions for multiple threads

To support multiple threads, SEI employs an array c of counters, one per thread (Algorithm 9); this array of counters is called the *barrier*. Before a thread starts executing the *Exec*$_1$ block, it increments its counter $c[i]$, and increments it again after completing the *Validate* block. Therefore, an odd counter indicates that a thread is executing event handling. Immediately after completing its *Validate* block, each thread reads c and makes sure that all other threads that executed traversals concurrently have finished checking their modifications to state s.

As we show in Algorithm 10, SEI intercepts lock operations to make sure that a thread keeps its locks throughout the first and second execution. Each thread adds its locks to a set Q and releases them only at the end of the check procedure.

5.3 Correctness with multiple threads

We are now ready to show the correctness of the full SEI algorithm with multiple threads. This boils down to showing that there is no error propagation among processes through shared variables. We now show that Theorem 2 holds even in presence of multiple threads.

Lemma 25 *If a set of threads $\tau_1, \ldots, \tau_n$ of a process π, with $n > 1$, are in the same critical section C while running traversal $E_1, \ldots, E_n$ respectively, then π crashes before all threads exit C.*

Proof:
1. Let L_C be the set of locks required to enter C.
2. Let s be the state when the last thread in $\tau_1, \ldots, \tau_n$ executes its first operation of C.
3. No fault occurs during traversal $E_1, \ldots, E_n$ after state s.
   3.1. Mutual exclusion is violated at state s.
   3.2. A fault has occurred before s in some traversal $E_1, \ldots, E_n$.
   3.3. Step 3 follows by fault frequency.
4. Each thread in $\tau_1, \ldots, \tau_n$ executes $Release(l)$ for each $l \in L_C$ before exiting C by definition of L_C and Step 3.
5. If some $l \in L_C$ is invalid, some thread crashes the process before exiting C, upon releasing l and checking that l is invalid by Steps 3 and 4.
6. Else, if all $l \in L_C$ are valid
   6.1. At most one thread holds all locks in L_C.
   6.2. Some thread τ_i crashes the process before exiting C, upon releasing some $l \in L_C$ and checking that τ_i does not hold l, by Steps 3 and 4. □

Lemma 26 *Let two threads τ_i and τ_j execute a block Exec$_1$ or Exec$_2$ concurrently in traversals E_i and E_j, respectively, after a fault occurs after the beginning of E_i or E_j. Assume block-confined faults (Assumptions 7). If the process does not crash before the end of E_i or E_j, then neither thread terminates its traversal before the other thread terminates executing the Exec$_1$ or Exec$_2$ block and checks all state modifications in its traversal in the Validate block.*

Proof:
1. No fault occurs during E_i or E_j in blocks different than *Exec*$_1$ or *Exec*$_2$ by fault frequency and block-confinedeness.

2. The entry of τ_i and τ_j in the counter vector c has been increased at state s_i and s_j during a fault-free period by Step 1.
3. $c[\tau_i]$ or $c[\tau_j]$ are valid at s_i and s_j because else the process crashes before the end of E_i or E_j by Step 2, contradiction.
4. $c[\tau_i]$ or $c[\tau_j]$ are set to an even value at s_i and s_j by Steps 2 and 3.
5. The *Validate* block of τ_i (wlog) completes only in the following cases:
   - 5.1. Case: $c_{snap}[j]$ is odd or it is even and $c_{snap}[j] < c[j]$
     - 5.1.1. τ_j has increased $c_{snap}[j]$ at some state $s \succ s_j$, by Step 4.
     - 5.1.2. State s can only occur after τ_j terminates the executing the *Exec*$_1$ or *Exec*$_2$ block and checks all state modifications in its traversal in the *Validate* block by Step 1.
   - 5.2. Case: $c_{snap}[j]$ is even and $\neg Valid(c[j])$
     - 5.2.1. Variable $c[j]$ is corrupted after the snapshot of c is taken, by Step 1 and since the validity of the variables in c is checked in Line +3 of Algorithm 9.
     - 5.2.2. The fault corrupting $c[j]$ must have occurred after E_i and E_j are terminated, by fault frequency. □

Lemma 27 *Theorem 2 holds in processes with multiple threads.*

Proof:
1. Two types of error propagation in multithreaded applications are not covered by Theorem 2.
2. Case: Mutual exclusion is violated.
   - 2.1. This is ruled out by Lemmas 25 and 26.
3. Case: A thread τ_i reads a corrupt variable v determined by an assignment from a different thread τ_j.
   - 3.1. Let τ_i be the first thread that reads a variable corrupted in this manner in a state s.
   - 3.2. Let E_j be the traversal in which τ_j last writes v before s.
   - 3.3. If τ_i has read v then τ_j has released the lock on v by Lemmas 25 and 26.
   - 3.4. Locks are only released after the *Validate* block.
   - 3.5. τ_j does not modify v after releasing the locks protecting v by definition of s.
   - 3.6. The value read by τ_i is the final value of v in E_j.
   - 3.7. Theorem 2 ensures that τ_i can detect the correctness of v. □

A corollary of Lemma 27 is the following.

Corollary 1 *Error isolation holds in presence of multiple threads.*

5.4 Discussion

We now discuss some limitations of our multithreading support in SEI and one possible alternative solution.

5.4.1 Limitations

Fault coverage. Our barrier mechanism assumes block-confined faults (see Lemma 26), an assumption that is not needed in the single-threaded case. The reason for that assumption can be illustrated with an example. An ASC fault can bring a thread τ_i to jump from $pc = 0$ directly inside a critical section CS. Assume τ_i is preempted after modifying some state variables inside CS. Next, another thread τ_j gets into CS and reads the corrupt variables. Thread τ_j can successfully acquire the locks protecting CS because τ_i skipped their acquisition due to the ASC fault. Moreover, τ_j does not block on the barrier at the end of the traversal because τ_i skipped the *Initialization* blocks altogether. Therefore, if τ_i is not scheduled before τ_j finishes its traversal, then τ_j may send corrupt messages and violate error isolation.

Block-confined faults rule out this scenario because a fault skipping the *Initialization* block cannot jump into an execution block. This assumption might be, however, unnecessarily strong. It would be sufficient to argue that a fault will not skip the *Initialization* block and jump inside an execution block and inside a critical section inside that block. Despite the slightly strong assumption, our experiments in the companion paper [3] do not reveal a major difference in the number of undetected errors between single-threaded and multithreaded executions.

Concurrency. Our algorithm implements a two-phase locking (2PL) scheme since locks are not released until the *Validate* block terminates. This locking protocol is known to limit the available parallelism under contended workloads. Two further issues with SEI can aggravate this problem. First, our algorithm increases critical section length by executing them twice. In our experiments with `memcached`, however, we have not experienced significant reduction in the system's throughput due to the locking scheme when enough parallelism is available (*i.e.*, when threads mostly access different keys). Second, the barrier approach can make fast threads wait for slow threads. That can be mitigated by "preempting" the traversal and letting the thread execute other work meanwhile – see the companion paper [3] for details.

Deadlocks. SEI targets applications where locks are acquired in a consistent order (see Assumption 8) because our 2PL approach might cause deadlocks otherwise. An alternative solution that does not require hierarchical locks is to rollback the state modifications (as *Reset* does) upon detecting a deadlock; and, subsequently, waiting for a random period of time before retrying. Note that the deadlock detection mechanism does not have to be harden because a misdetection only affects liveness but not safety.

5.4.2 The mini-traversals approach

Given the limitations of the 2PL scheme, one might consider the following alternative solution. We split a traversal E into mini-traversals by considering every unlock event as an externalization event. Indeed, once a thread τ_i releases a lock l, the data protected by l can be read by another thread τ_j, which in turn can use the data to create and send a message; hence, unlocking can be seen as state externalization.

The mini-traversal approach can be realized by wrapping lock release functions. Whenever a release function is called in a traversal E, before the lock is actually released, the traversal is reset, re-executed, and checked (*Reset*, $Exec_2$, and *Validate* blocks). After that, the traversal sends a local message, effectively splitting the traversal E into two mini-traversals E_1 and E_2. The local message represents the local state (including acquired locks) being transferred from mini-traversal E_1, which has just finished, to mini-traversal E_2, which is about to start.

We have implemented this solution, but practical limitations made it very inefficient. In particular, since traversals are split, a mini-traversal E_i might start at deep level in the call stack and end at a higher level in the call stack. If that occurs, all stack frames between both stack levels have to be reset before the traversal E_i can start its $Exec_2$ block. The need to copy large portions of the stack after every lock release render the mini-traversals approach hard to employ in practice; for example, in `memcached` more than 10 lock releases might occur per traversal, each copying as much as 800 bytes of stack data.

6 Additional experimental results

We present here some additional evaluation material.

6.1 Software fault injection

6.1.1 Setup and methodology

In our fault injection experiments, we follow the approach of Basile *et al.* [2] and Correia *et al.* [9] injecting single bit flips. For our fault injection experiments, we have implemented BFI[4], a tool for Intel's Pin dynamic binary instrumentation framework [24] that can inject faults during run-time. BFI can inject three groups of faults described in Table 4.

A control-flow (CF) fault flips a bit of the instruction pointer. A fault in the data-flow (DF) group affects the computation: WREG and WVAL represent incorrectly computed values that are respectively written into a register or a memory location, *e.g.*, an addition that results in a wrong value and is stored in a register; WADDR and RADDR represent computational errors while calculating an indexed address for reading or writing from memory. Finally, a fault in the RD group directly corrupts a register (RREG) before being used or a memory location (RVAL) before being read.

Field studies show that most memory faults are detected by ECC [18, 29]. Injected RVAL faults, however, automatically overwrite both, the value and its ECC. Hence, RVAL faults represent worst-case scenarios in which the ECC memory is not able to detect data corruption as assumed by corruption coverage (see Section 4.1).

[4] BFI stands for bit-flip injector.

Group	Fault	Description
CF	CF	IP register changes (control-flow fault)
DF	WREG	register value changes after it is written
	WVAL	memory value changes after it is written
	WADDR	calculated address changes before write
	RADDR	calculated address changes before read
RD	RREG	register value changes before it is read
	RVAL	memory value changes before it is read

Table 4: Fault types supported by BFI.

To speed up our experiments and make the results reproducible, we have modified `memcached` to read commands from an input-trace file and write responses into an output-trace file by wrapping functions reading from and writing to sockets. To compare the output trace, we first create a golden run output-trace file. We perform two sets of experiments. The first set studies the fault coverage of SEI and the effects of leveraging hardware error detection codes in the implementation. We run, with a single thread, the unhardened `memcached` (mc), the SEI-hardened variant (mc-sei) with hardware error detection codes, and a further SEI-hardened variant (mc-sei-dup) with duplicated state assuming no error detection codes in hardware. The second set of experiments investigates whether the computational scalability aspect of our implementation affects the fault coverage. In this set, we run mc-sei and mc-seil with 4 threads; mc-seil has the checking barrier disabled and assumes that locks are not skipped. We perform 8,000 executions for each fault type and each single-threaded variant, with a subtotal of 96,000 executions for the DF group, 48,000 for the RD group, and a total of 168,000 executions (see Table 5). For the multithreaded experiments, we perform a total of 80,000 executions (see our companion paper [3]).

Each fault injection execution consists of three phases. A *warmup phase*, where set commands are issued to populate the cache, but no faults are injected; an *injection phase*, where set and get commands are issued and one fault is injected at a randomly selected instruction; and finally, a *propagation phase*, where all keys are retrieved multiple times with get commands, but again no faults are injected. Note that some instructions are not susceptible to every fault, for example, an instruction that does not write to memory cannot suffer a WADDR fault. In such cases, we inject the fault in the first susceptible instruction after the selected one. Moreover, if multiple registers/addresses operands are susceptible to the fault then the operand is selected randomly.

The output-trace of each execution is compared with the golden run. In each run, one fault is injected at a randomly selected instruction inside or outside the event handler including shared libraries; Pin cannot, however, instrument instructions inside syscalls. A fault that causes a trace deviation, *e.g.*, an unexpected message or a shorter trace, produces a *manifested error*. The errors we report are all manifested, consequently we refer to them as just errors henceforth.

6.1.2 Results with `memcached`

Table 5 summarizes the main results of our fault injection experiments *broken down by fault type.* The right-most column shows, for each fault-variant combination, the total number of manifested errors out of 8000 injections. Manifested errors are classified in detected and undetected, shown as percentage of the total number of manifested errors. *Undetected* errors are *corrupt* output messages that cannot be detected by the client. They correspond to error propagation scenarios where the error isolation property is violated. Detected errors are further divided into *det/SEI*, *i.e.*, errors detected and isolated by `libsei`, for example, crashes initiated by the library or invalid messages detectable at the client; and *det/other* errors, *i.e.*, errors detected or isolated by other mechanisms, for example, crashes due to segmentation fault or assertions, infinite loops, and also error messages or partial messages detectable at the client.

The most important result of our fault injection experiments is the drastic decrease of undetected errors when hardening `memcached`. Aggregating all results, mc shows 33.43% undetected errors while mc-sei only 0.25%. Undetected errors in the mc variant range from 9% up to 69% of the manifested errors, depending on the fault type. In contrast, the mc-sei variant shows at most 0.83% undetected errors. Table 5 also shows that `libsei` detects and isolates from 14% up to 82% of the manifested errors (47.04% aggregated).

Fault	Variant	Errors			
		Undetected	Det/SEI	Det/other	Total
CF	mc	9.66%	-	90.34%	6690
	mc-sei	0.06%	14.70%	85.23%	6515
	mc-sei-dup	0.00%	9.87%	90.13%	6594
WREG	mc	34.50%	-	65.50%	4194
	mc-sei	0.13%	52.93%	46.93%	4481
	mc-sei-dup	0.00%	40.41%	59.59%	4180
WVAL	mc	69.92%	-	30.08%	3304
	mc-sei	0.16%	82.26%	17.58%	5063
	mc-sei-dup	0.00%	79.80%	20.20%	2510
WADDR	mc	45.51%	-	54.49%	3564
	mc-sei	0.11%	61.20%	38.69%	5412
	mc-sei-dup	0.00%	46.88%	53.12%	4394
RADDR	mc	32.25%	-	67.75%	4118
	mc-sei	0.21%	34.35%	65.54%	5297
	mc-sei-dup	0.00%	32.06%	67.94%	4907
RREG	mc	25.55%	-	74.45%	5678
	mc-sei	0.21%	39.49%	60.30%	5700
	mc-sei-dup	0.00%	34.09%	65.90%	5453
RVAL	mc	41.65%	-	58.35%	4936
	mc-sei	0.83%	53.97%	45.20%	5803
	mc-sei-dup	0.00%	62.83%	37.17%	5989
Aggregate	mc	33.43%	-	66.57%	32484
	mc-sei	0.25%	47.04%	52.71%	38271
	mc-sei-dup	0.00%	43.35%	56.64%	30363

Table 5: Errors classified in undetected, SEI-detected, and detected with other mechanisms. Total errors out of 8000 executions for each fault-variant combination.

Fault	Variant	Errors			
		Undetected	Det/SEI	Det/other	Total
CF	dw	8.71%	-	91.29%	3251
	dw-sei	0.12%	9.33%	90.55%	3334
WREG	dw	26.41%	-	73.59%	1946
	dw-sei	0.09%	36.45%	63.46%	2140
WVAL	dw	35.95%	-	54.05%	1235
	dw-sei	0.06%	56.54%	43.40%	1567
WADDR	dw	27.50%	-	72.50%	1818
	dw-sei	0.28%	39.31%	60.40%	2139
RADDR	dw	40.14%	-	59.86%	2070
	dw-sei	0.04%	41.42%	58.54%	2501
RREG	dw	23.01%	-	76.99%	2686
	dw-sei	0.07%	24.88%	75.05%	3014
RVAL	dw	44.37%	-	55.63%	2565
	dw-sei	0.53%	41.84%	57.63%	2818
Aggregate	dw	27.80%	-	72.20%	15571
	dw-sei	0.18%	33.02%	66.80%	17513

Table 6: Errors classified in undetected, SEI-detected, and detected with other mechanisms. Total errors out of 4000 executions for each fault-variant combination.

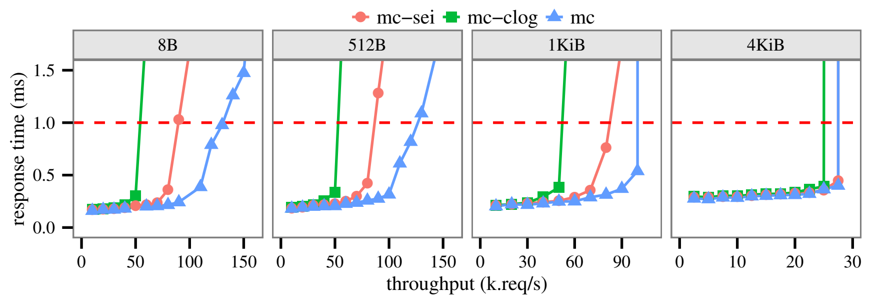


Figure 10: Response time versus throughput varying value size. At the 1 ms cut mc-sei achieves 70% of the throughput of mc with 8 B values and 80% with 1 KiB values.

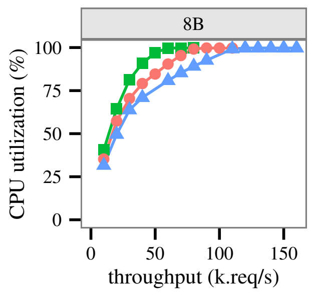


Figure 11: CPU utilization versus throughput with 8 B values

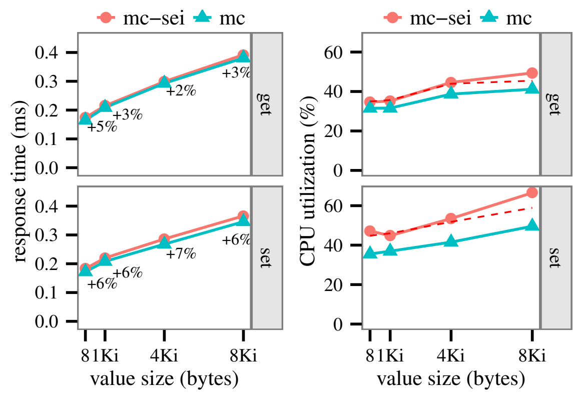


Figure 12: Response time and CPU utilization varying message size for gets and sets with 10 k.req/s load

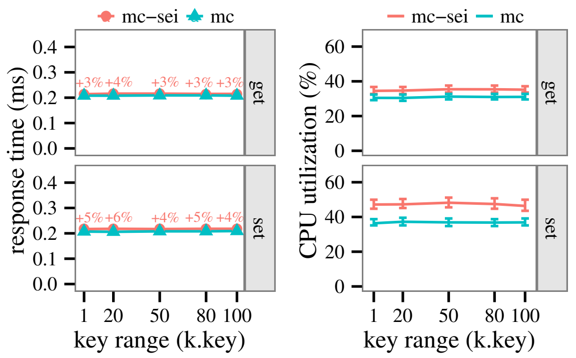


Figure 13: Response time and CPU utilization varying number of keys with 10 k.req/s load (value size 1KiB)

6.1.3 Results with Deadwood

We performed fault injection experiments with Deadwood injecting 4000 faults of each fault type. As with `memcached`, we modified Deadwood to read input messages from an input-trace file and store the outgoing messages to an output file. We followed the same three-phase approach: the cache of the resolver is populated with DNS records in the warmup phase without injecting faults; in the injection phase the resolver is queried for the records and faults are injected; no faults are injected in the propagation phase while reading from the cache.

Table 6 summarizes the results of our fault injection experiments with Deadwood broken down by fault type. We observe high decrease in the number of undetected errors, from 27.80% in the native version down to 0.18% in the hardened when aggregated over all fault types; and from 32.38% in the native version down to 0.12% in the hardened when aggregated over DF faults only.

6.2 Single-thread performance of `memcached`

`libsei` is designed to amortize its performance overhead with a large number of threads. However, even in a single threaded configuration, it has 1.6x higher throughput than our PASC adaptation, mc-clog, and a viable overhead compared to mc. The value sizes range from extremely small messages (8 bytes) to fairly large message (4 kilobytes).

Figure 10 shows the response time versus throughput for `memcached` using different messages sizes and varying the load of get operations. Note that the vast majority of the operations in typical workloads are gets [1, 27, 30]. For 1 KiB and 4 KiB large values, the response time elbow of mc raises at the limit of the network indicating that mc is network bound. For smaller value sizes, mc is CPU bound. With 4 KiB large values, mc-sei also saturates the network, while being CPU bound with 1 KiB or smaller values. In contrast, mc-clog is CPU bound for all measured value sizes.

Figure 11 shows the CPU utilization for the experiments with 8 B value size. As expected, all variants have a response time elbow once the CPU utilization reaches 100%, *i.e.*, at about 55, 80 and 105 k.req/s for mc-clog, mc-sei and mc, respectively. The throughput increases even though the CPU has reached its limit due to batching at the socket level. Response times above 1 ms are, however, not desired in systems such as memcached mainly used for speeding up database queries. Hence, we define memcached's capacity at 1 ms response time, where mc-sei has 70% of the mc's throughput with 8 B large values and about 80% with 1 KiB large values, representing 30% and 20% overhead, respectively. The overhead of mc-clog is at least two times larger, *i.e.*, about 58% with 8 B values and 48% for 1 KiB. Presenting a substantially higher overhead than mc-sei, we do not evaluate mc-clog any further.

We now investigate the influence of the value size on the overhead of get and set operations. Figure 12 depicts the response time and CPU utilization of mc-sei and mc varying the value size from 8 B to 8 KiB with a key range of 1000 keys and a load of 10 k.req/s. The value size increases the message size and, consequently, the response time. The difference of response time varies from 2% to 7% (small labels on the top of the response time measurements). Larger value sizes also affect the CPU utilization because larger messages have to be copied from/to the socket. Furthermore, mc-sei's CPU utilization overhead increases with the value size irrespective of whether get or set operations are issued. For example, for set requests, the difference of CPU utilization between mc-sei and mc increases from 7.8% up to 17%. The reason can be tracked down to the CRC calculations: The dashed lines show the average CPU utilization of mc-sei with CRC calculation disabled. The difference between the dashed line and mc is roughly constant around 9%. Large requests, however, saturate the network before CPU (see 4 KiB, Figure 10), so the additional CRC computation does not cause any important performance penalty. Also, many practical workloads are comprised of small requests. For such workloads, this additional CRC computation is not an issue.

We now investigate the influence of the key range on mc-sei and mc varying the key range from 1000 to 100,000 keys with 1 KiB large values and a load of 10 k.req/s. Note that the unit of the x-axis is 1000 keys. On the top of the response time measurements a small label indicates the overhead of mc-sei. No trend can be identified in the response time or CPU utilization with the increasing number of keys. The response time overhead varying between 3% and 6%. The CPU utilization *difference* is about 4% and 11% for get and set operations, respectively. The additional overhead for set operations is expected due to the longer code path executed in set operations.

6.3 Single-thread performance of Deadwood

Deadwood is a single-threaded server and only evaluated in the "single-thread scenario" section.

6.3.1 Setup and methodology

Experiments with Deadwood follow a similar setup, but with up to 20 client machines running nsping - a DNS querying tool available in BSD ports. Clients send requests to resolve a domain name randomly selected from the list of 100 most visited websites (using data from http://www.alexa.com). In the warm-up phase the resolver's cache is empty, hence a first request for every domain name is forwarded to an upstream server. Once the upstream server replies, the response is cached and sent to the client. Further requests for the same domain name are served from the cache. CRC checksums for the outgoing messages are calculated, but not sent out.

6.3.2 Results

Similar to memcached, we investigate the performance of hardened version of Deadwood compared to native Deadwood. Due to rather small message sizes (average size of a request message is 28 B, response - 76 B) both variants of Deadwood are CPU bound.

Figures 14 and 15 show response time versus throughput under varying load. dw-sei reaches CPU bound faster due to double execution of event handlers, at approximately 33 k.reg/s, while dw achieves 53 k.req/s. This is consistent with CPU utilization measurements, presented in Figure 16. The overhead of dw-sei is thus 38%.

Figure 14 shows the response time before reaching 100% CPU consumption: under moderate load the overhead varies between 8% and 13%.

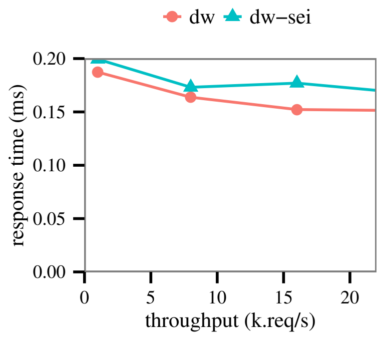


Figure 14: Response time versus throughput before reaching 100% CPU utilization

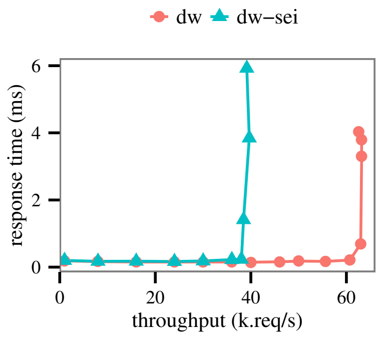


Figure 15: Response time versus throughput

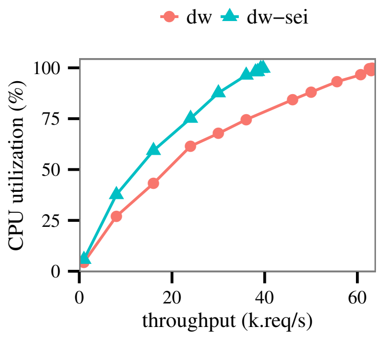


Figure 16: CPU utilization versus throughput

7 Software artifacts

The following software artifacts are available as open source:

Artifact	**URL**
`libsei`	http://bitbucket.org/db7/libsei
Hardened `memcached`	http://bitbucket.org/db7/libsei-memcached
Hardened Deadwood	http://bitbucket.org/db7/libsei-deadwood
Bit-flip injector (BFI)	http://bitbucket.org/db7/bfi

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Workload analysis of a large-scale key-value store”. In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems.* (London, England, UK). SIGMETRICS ’12. New York, NY, USA: ACM, 2012, pp. 53–64. ISBN: 978-1-4503-1097-0. DOI: 10.1145/2254756.2254766.

[2] C. Basile, Long Wang, Z. Kalbarczyk, and R. Iyer. “Group communication protocols under errors”. In: *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on.* 2003, pp. 35–44. DOI: 10.1109/RELDIS.2003.1238053.

[3] Diogo Behrens, Marco Serafini, Flavio P. Junqueira, Sergei Arnautov, and Christof Fetzer. “Scalable Error Isolation for Distributed Systems”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* Oakland, CA: USENIX Association, May 2015. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/behrens.

[4] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer, 2011.

[5] Miguel Castro and Barbara Liskov. “Practical Byzantine fault tolerance”. In: *Proceedings of the third symposium on Operating systems design and implementation.* (New Orleans, Louisiana, United States). OSDI ’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://portal.acm.org/citation.cfm?id=296806.296824.

[6] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. “Attested append-only memory: making adversaries stick to their word”. In: *SIGOPS Oper. Syst. Rev.* 41 (6 Oct. 2007), pp. 189–204. ISSN: 0163-5980. DOI: 10.1145/1323293.1294280.

[7] B.A. Coan. “A compiler that increases the fault tolerance of asynchronous protocols”. In: *Computers, IEEE Transactions on* 37.12 (Dec. 1988), pp. 1541–1553. ISSN: 0018-9340. DOI: 10.1109/12.9732.

[8] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. *Models and Algorithms for ASC Hardening and a Correctness Proof.* YL-2011-003. Yahoo! Labs, 2011.

[9] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. "Practical Hardening of Crash-tolerant Systems". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. (Boston, MA). USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=2342821.2342862.

[10] Flavin Cristian. "Understanding fault-tolerant distributed systems". In: *Commun. ACM* 34.2 (Feb. 1991), pp. 56–78. ISSN: 0001-0782. DOI: 10.1145/102792.102801.

[11] Flaviu Cristian. "A Rigorous Approach to Fault-Tolerant Programming". In: *Software Engineering, IEEE Transactions on* SE-11.1 (Jan. 1985), pp. 23–31. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.231534.

[12] Flaviu Cristian and Christof Fetzer. "The Timed Asynchronous Distributed System Model". In: *IEEE Trans. Parallel Distrib. Syst.* 10 (6 June 1999), pp. 642–657. ISSN: 1045-9219. DOI: 10.1109/71.774912.

[13] Amazon Web Services Service Health Dashboard. *Amazon S3 Availability Event: July 20, 2008*. http://status.aws.amazon.com/s3-20080720.html. July 2008.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *J. ACM* 32 (2 Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121.

[15] Felix C. Gärtner. "Fundamentals of fault-tolerant distributed computing in asynchronous environments". In: *ACM Comput. Surv.* 31.1 (Mar. 1999), pp. 1–26. ISSN: 0360-0300. DOI: 10.1145/311531.311532.

[16] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. "Characterization of linux kernel behavior under errors". In: *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*. 2003, pp. 459–468. DOI: 10.1109/DSN.2003.1209956.

[17] Marc Hamilton. *Software development: building reliable systems*. Prentice Hall Professional, 1999.

[18] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. (London, England, UK). ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150989.

[19] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. "Byzantine Fault Detectors for Solving Consensus." In: *Comput. J.* (2003), pp. 16–35.

[20] Leslie Lamport. *Specifying systems*. Addison-Wesley Reading, 2002.

[21] Leslie Lamport and Stephan Merz. "Specifying and Verifying Fault-Tolerant Systems". In: *Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*. ProCoS. London, UK, UK: Springer-Verlag, 1994, pp. 41–76. ISBN: 3-540-58468-4. URL: http://dl.acm.org/citation.cfm?id=646843.706634.

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4 (3 July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[23] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. "TrInc: small trusted hardware for large distributed systems". In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. (Boston, Massachusetts). NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–14. URL: http://dl.acm.org/citation.cfm?id=1558977.1558978.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. (Chicago, IL, USA). PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034.

[25] Gil Neiger and Sam Toueg. "Automatically increasing the fault-tolerance of distributed systems". In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. (Toronto, Ontario, Canada). PODC '88. New York, NY, USA: ACM, 1988, pp. 248–262. ISBN: 0-89791-277-2. DOI: 10.1145/62546.62588.

[26] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs". In: *Proceedings of the sixth conference on Computer systems.* (Salzburg, Austria). EuroSys '11. New York, NY, USA: ACM, 2011, pp. 343–356. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966477.

[27] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation.* (Lombard, IL). nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 385–398. URL: http://dl.acm.org/citation.cfm?id=2482626.2482663.

[28] D. Powell. "Failure mode assumptions and assumption coverage". In: *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on.* July 1992, pp. 386 –395. DOI: 10.1109/FTCS.1992.243562.

[29] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM errors in the wild: a large-scale field study". In: *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems.* (Seattle, WA, USA). SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 193–204. ISBN: 978-1-60558-511-6. DOI: 10.1145/1555349.1555372.

[30] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. "TAO: how facebook serves the social graph". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* (Scottsdale, Arizona, USA). SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 791–792. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213957.

[31] Timo Warns. *Structural Failure Models for Fault-Tolerant Distributed Computing.* Springer, 2010.

[32] Henry S. Warren. *Hacker's Delight.* 2nd. Addison-Wesley Professional, 2012.

[33] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1-55860-508-8.