Formalising Fault Injection and Countermeasures

Thomas Given-Wilson and Axel Legay
thomas.given-wilson,axel.legay@uclouvain.be
Université Catholique de Louvain
Belgium

ABSTRACT

Fault injection is widely used as a method to evaluate the robustness and security of a system against many kinds of faults and attacks. Recent works have considered many ways to demonstrate security risks and viable attacks using fault injection, and some have also proposed countermeasures. However, no general and formal definition of fault injection or countermeasure has been provided that can be used to reason about such attacks. This leaves significant results in this area to be ad-hoc and without broad applicability. This paper presents formal definitions of both fault injection on an arbitrary system and what an effective countermeasure is. These definitions are used to prove that fault injection attacks *cannot* in general be prevented (by any countermeasure). An example is presented that demonstrates how to construct an effective countermeasure for a specific fault injection that parallels some well known approaches. Further extensions to account for probabilistic behaviour and systems with time are also presented. These definitions and results demonstrate formal proofs about the security and defences of systems in ways that can be used, thus yielding a broadly applicable approach that can formalise fault injections and countermeasures in the future.

ACM Reference Format:
Thomas Given-Wilson and Axel Legay. 2020. Formalising Fault Injection and Countermeasures. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3407023.3407049

1 INTRODUCTION

Fault injection has been widely and increasingly used both to test the robustness of software systems, and as a method to attack such software systems. There are a wide variety of different kinds of fault injections that can trigger unwanted or malicious behaviour. Many attacks have been demonstrated on various systems, showing that faults can be injected into many kinds of devices and achieve many kinds of faults or attacks [1, 7, 8, 10, 12, 19, 20, 28, 33, 34]. This also includes some recent examples of attacks that can be achieved using only software, an example of this is row hammer [16] that has been exploited in various attacks [24, 29, 36]. Thus the current state-of-the-art is that many new fault injections are being found, and indeed many new approaches to automatically discover fault injection vulnerabilities [10, 11].

Due to this large number of fault injection vulnerabilities that are continuing to be found, some recent works have attempted to implement countermeasures that can prevent a fault injection causing harm [2, 3, 7, 17, 21, 26]. The goal of these works is to demonstrate that *some* fault injection attacks or other vulnerabilities can be mitigated by some kind of countermeasure.

More broadly, both the definition of a fault injection (attack) and what it means to be a countermeasure to such a fault injection attack has been treated in an ad-hoc manner [1, 7, 8, 10–12, 16, 19, 20, 24, 28, 29, 33, 34, 36]. That is, no general definition that can be formally reasoned about has been provided. This makes it infeasible to clearly define the limits of what a fault injection attack is, and whether a countermeasure is truly effective or not.

This paper approaches the challenges of defining what a fault injection is for an arbitrary computing system by considering the foundations of computation. By defining fault injection on *Turing Machines* (TM) and *Universal Turing Machines* (UTM) [14, 15, 31, 32] it is possible to define how a fault operates on an arbitrary computer. In addition to being well recognised key definitions of computability, including computation of other machines and programs, the representation of TMs and UTMs captures key distinctions of the components of the von Neumann architectures used in almost all contemporary systems [18, 22].

Fault injections can be defined as a *Parallel Turing Machine* (PTM) [13, 25, 35] where a single tape is shared between two (or more) TMs that may both operate on this tape. One TM is defined to be the normal behaviour of the program, and a second TM to be the behaviour of injecting faults via the tape. This abstraction allows for a very simple and straightforward representation of the interesting behaviour, as well as having a strong mathematical basis to work with.

Combining these concepts allows the clear definition of the program and the fault injection upon that program. This formalises both the capabilities of a fault injection and their affects on the program. Further, an *effective countermeasure* can be defined: an equivalent program that prevents a fault injection from changing the original program's behaviour.

Armed with this formalism we then prove that equivalent behaviour is impossible to protect from arbitrary fault injection. That is, fault injection attacks *cannot* in general be prevented (by any possible countermeasure) from changing program behaviour. The intuition behind this result is to consider the output of the program. This output can be changed by fault injection, and so the program behaviour changed. The only way to prevent this, is to check the output before termination. However the fault injection can always occur after this check, or inside the check itself. The only resolution is a program that never terminate and produces output, which

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3407049

cannot be equivalent to a program that terminates and produces output.

This result may be unsurprising since an unbounded fault injection attack is unbounded in the damage it can cause to the system. However, the fault models necessary to prove an effective countermeasure impossible prove very simple to implement in practice.

More interesting is to consider under which definitions of equivalent, and for which fault models, an effective countermeasure *can be defined*. This paper also presents an example of an effective countermeasure using redundancy techniques similar to current systems. This demonstrates how to exploit the formalism here to construct a proof of an effective countermeasure in practice.

These results are then extended with probabilistic behaviour and time to demonstrate broader flexibility and utility of the approaches defined here.

Probabilistic systems allow the representation of the probability of a fault occurring, and also definition of a countermeasure can be effective with a high degree of confidence. This allows strong results to be achieved to demonstrate the robustness of a system when the probability of a fault can be approximated. The intuition here is that given the probability of a fault occurring, it is possible to define a countermeasure that will ensure the system's behaviour remains unchanged with very high probability.

Systems with concepts of time allow for the representation of faults that occur infrequently and for reasoning over the timing of program execution. In practice this allows results that show an effective countermeasure against fault injections that are rate limited, or when the program has time to recover between faults.

The contributions of the paper are as follows.

- Defining a formal model for fault injection attacks upon systems (defined using TMs and variations).
- Formally defining an effective countermeasure for the above model.
- Proving the impossibility of an effective countermeasure against arbitrary fault injection attacks.
- Proving the impossibility of an effective countermeasure up to output equivalence against even a very limited fault injection attack.
- Demonstrating how to prove a countermeasure to be effective, including an example.
- Demonstrating how to prove probabilistic results, including probabilistically effective countermeasures.
- Demonstrating how to account for faults over time, and prove countermeasures against rate-limited faults.

The structure of the paper is as follows. Section 2 recalls background on various forms of Turing Machines. Section 3 defines the model of a system and fault injection attacks. Section 4 formalises what it means to be an effective countermeasure and proves they are in general impossible. Section 5 demonstrates how to formalise and prove an effective countermeasure. Section 6 considers variations and extensions to the results. Section 7 discusses the results and their relation to real-world applications. Section 8 concludes.

2 BACKGROUND

This section recalls various Turing Machines as a formalism to represent an arbitrary computing system.

2.1 Turing Machines

This section recalls *Turing Machines* as a universal model of computation [4, 31, 32]. The majority of this section is notation, as key results rely upon the definition of TMs.

Assume an *alphabet* $\mathcal{A}$ to be a finite set of *symbols* $a \in \mathcal{A}$. Denote by $|\mathcal{A}|$ to be the number of symbols in $\mathcal{A}$, e.g., if $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_i\}$ then $|\mathcal{A}| = i{+}1$. Observe that for any alphabet the symbols of that alphabet can be given an indexing based upon the natural numbers. Here $|\mathcal{A}|$ is assumed to be greater than 1, i.e., there are at least two symbols in the alphabet. This is by convention since TMs are normally defined to have a blank symbol b and at least one other symbol is needed to be able to make use of symbols on the tape.

A *tape* $\mathcal{P}$ is an infinite sequence of symbols $a_0, a_1, a_2, \ldots$ indexed by the natural numbers. Denote by $i\#\mathcal{P}$ the ith symbol of the tape $\mathcal{P}$, e.g., $1\#\mathcal{P}$ where $\mathcal{P} = a_0, a_1, a_2, \ldots$ denotes a_1 and denote by $\mathcal{P}[i := a]$ the tape $\mathcal{P}$ with the ith symbol set to a.

Aside: The traditional definition of TMs assumes an infinite tape in both directions, instead of here only infinite in one direction. There are straightforward approaches to adapt a tape that is infinite in both directions to a single direction for a TM, here this is assumed a priori for simplicity.

When a TM operates on a tape, the TM has a current position on the tape. Denote by $pos_{\tau}(\mathcal{P})$ the current position on the tape $\mathcal{P}$ according to the TM $\mathcal{T}$. When no ambiguity may occur, the notation $pos(\mathcal{P})$ is used.

TMs exploit a set of movements they can perform with respect to the current position on their tape. To support this assume a finite set of *movements* $m \in \mathcal{M}$ that are functions $f : \mathbb{N} \to \mathbb{N}$ where $\mathbb{N}$ is the natural numbers. Traditionally there are two movements: *left* that would be the function that decrements the current position by one; and *right* that increments the current position by one. The choice to use functions on natural numbers here is to allow for movements that are more complex than a single step left or right, and also to simplify edge cases when a movement would otherwise extend below the 0th position on the tape (i.e., when moving left from position 0).

Also assume a finite set of *states* $s \in \mathcal{S}$. These represent the possible states of the TM and include an initial state s_I and at least one final state s_F.

Definition 2.1 (Turing Machine). A Turing Machine $\mathcal{T}$ is a tuple $(\mathcal{S}, \mathcal{A}, \to, s_c, \mathcal{P})$ with: states $\mathcal{S}$ including an initial state s_I and at least one final state s_F; alphabet $\mathcal{A}$; a *transition relation* $\to$; a *current state* s_c; and tape $\mathcal{P}$. The transition relation $\to$ is a set of tuples (s_w, a_x, a_y, m, s_z) where s_w and s_z are states, a_x and a_y are symbols, and m is a movement.

Intuitively the tuples (s_w, a_x, a_y, m, s_z) of the transition relation indicate that when $\mathcal{T}$ is in a state s_w and reads the symbol a_x then $\mathcal{T}$ write the symbol a_y at the current position, performs the movement m, and transitions to the state s_z. Formally, a TM $\mathcal{T}$ transitions to a TM $\mathcal{T}'$ (denoted $\mathcal{T} \mapsto \mathcal{T}'$) when $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \to$

$, s_w, \mathcal{P})$ and $\mathcal{T}' = (\mathcal{S}, \mathcal{A}, \rightarrow, s_z, \mathcal{P}')$ and $(s_w, a_x, a_y, m, s_z) \in \rightarrow$ and $\mathcal{P}' = \mathcal{P}[pos(\mathcal{P}) := a_y]$ and $pos(\mathcal{P}') = m(pos(\mathcal{P}))$. Denote with $\mapsto\!\!\!\!\!\!\mapsto$ the reflexive transitive closure of $\mapsto$, and by $\perp$ an infinite sequence of reductions $\mapsto\!\!\mapsto \ldots$.

The initial state s_I is the first state the TM will being in and exists by convention. The final states s_F are those states that does not appear at the beginning of any transition, i.e., there exists no tuples of the form (s_F, a_x, a_y, m, s_z) in $\rightarrow$ for any a_x, a_y, m, or s_z.

Note that it is possible to define the transition relation such that two (or more) tuples $(s_{w1}, a_{x1}, a_{y1}, m_1, s_{z1})$ and $(s_{w2}, a_{x2}, a_{y2}, m_2, s_{z2})$ may exist where $s_{w1} = s_{w2}$ and $a_{x1} = a_{x2}$ (where the rest of a_{y1}, a_{y2} and m_1, m_2 and s_{z1}, s_{z2} may or may not be equal). In practice such transition relations allow for non-deterministic TMs.

Observe that given a TM $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, s_c, \mathcal{P})$ that $\mathcal{S}$ and $\mathcal{A}$ and $\rightarrow$ remain fixed under reduction. That is, when $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, s_c, \mathcal{P})$ and $\mathcal{T} \mapsto \mathcal{T}'$ and $\mathcal{T}' = (\mathcal{S}', \mathcal{A}', \rightarrow', s_c', \mathcal{P}')$ then $\mathcal{S} = \mathcal{S}'$ and $\mathcal{A} = \mathcal{A}'$ and $\rightarrow = \rightarrow'$. Thus for simplicity denote with $\mathcal{T}(\cdot_1, \cdot_2)$ the TM $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, \cdot_1, \cdot_2)$ with current state $\cdot_1$ and tape $\cdot_2$.

Definition 2.2 (Halted). A Turing Machine $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, s_c, \mathcal{P})$ is *halted* (alternatively *terminated*) if there exists no tuple (s_c, a_x, a_y, m, s_z) in $\rightarrow$ where $a_x = pos(\mathcal{P})$ (for any a_y and m and s_z).

2.2 Universal Turing Machines

A useful capability of TMs is the ability to simulate another TM, this is done by a *Universal Turing Machine* (UTM). The UTM takes the representation of another TM $\mathcal{T}$ encoded onto the UTM's tape, and simulates $\mathcal{T}$. The details of how this is done are not recalled here (for details see [14, 15]), the key point to know is that this encodes an entire TM onto the tape of the UTM, and so any aspect of the encoded TM can be observed or modified directly through the tape of the UTM.

Definition 2.3 (Universal Turing Machine Representation). Denote by $\overline{\mathcal{T}}$ the Universal Turing Machine that simulates the Turing Machine $\mathcal{T}$.

The choice of TMs and UTMs is to be extremely general, while also having a notion of the concepts in modern computing systems (i.e., von Neumann architectures as implementing modern hardware). One key benefit is to be able to distinguish: the instructions part of a program (represented by the TM tape); and the hardware functions such as the CPU (represented by the transition function of the TM). This is particularly useful when considering the UTM, since this mirrors current systems when the entire program is written into memory (a UTM tape) and interpreted by hardware (the UTM current state and transition relation).

2.3 Parallel Turing Machines

One extension of TMs is *Parallel Turing Machines* (PTM) that support multiple TMs operating on a single shared tape [13, 25, 35]. The main interest in the design of PTMs was to allow the representation of parallel computing where multiple computations can share the same memory (or tape).

One of the main challenges with PTMs is in the synchronisation and safety between the two (or more, although this work only considers two) TMs that share the same tape. For parallel computing this is a major challenge to address, however for fault injection and countermeasures as considered here the most permissive (or dangerous) behaviour is allowed. That is, it is assumed that the two TMs sharing the tape may behave in a manner that is disruptive or dangerous to each other and share no safety of synchronisation mechanisms designed to prevent "bad" behaviour.

Definition 2.4 (Parallel Turing Machines). Given two Turing Machines $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{T}_1 = (\mathcal{S}_1, \mathcal{A}_1, \rightarrow_1, s_{w1}, \mathcal{P}_1)$ and $\mathcal{T}_2 = (\mathcal{S}_2, \mathcal{A}_2, \rightarrow_2, s_{w2}, \mathcal{P}_2)$ such that $\mathcal{A}_1 = \mathcal{A}_2$ and $\mathcal{P}_1 = \mathcal{P}_2$ then the *Parallel Turing Machine* $\mathcal{T}_1 \oplus \mathcal{T}_2$ that allows both $\mathcal{T}_1$ and $\mathcal{T}_2$ to share the same tape.

Observe that for clarity this paper shall consider interleaving behaviour for the parallel operations of PTMs, although none of the results here rely upon this choice.

Intuitively the behaviour of the PTM $\mathcal{T}_1 \oplus \mathcal{T}_2$ can be considered to be of both $\mathcal{T}_1$ and $\mathcal{T}_2$ operating independently except that each alteration of the tape affects the other, i.e.:

- when $\mathcal{T}_1 \mapsto \mathcal{T}_1'$ of the form $\mathcal{T}_1 = \mathcal{T}_1(s_{w1}, \mathcal{P}) \mapsto \mathcal{T}_1(s'_{w1}, \mathcal{P}')$ then this induces that $\mathcal{T}_2 = \mathcal{T}_2(s_{w2}, \mathcal{P})$ becomes $\mathcal{T}_2(s_{w2}, \mathcal{P}')$, and
- when $\mathcal{T}_2 \mapsto \mathcal{T}_2'$ of the form $\mathcal{T}_2 = \mathcal{T}_1(s_{w2}, \mathcal{P}) \mapsto \mathcal{T}_2(s'_{w2}, \mathcal{P}')$ then this induces that $\mathcal{T}_1 = \mathcal{T}_1(s_{w1}, \mathcal{P})$ becomes $\mathcal{T}_1(s_{w1}, \mathcal{P}')$.

In particular the other TM that shares the tape $\mathcal{P}$ has the same change made to the tape, although this is not a transition (or reduction) of the other TM.

Note that PTMs can introduce non-determinism and interactions between the TM that neither can exhibit alone. In this work this will be exploited to have on TM be the normal behaviour of the system, and the other parallel component of the PTM act as the fault injection mechanism.

Finally, observe that Parallel TMs do not have any inherent communication or synchronisation mechanisms outside of the shared tape. In particular there is no way for one TM to determine the status of another TM that shares the same tape outside of sending messages via the shared tape itself. In particular two TMs sharing the same tape cannot determine the state or termination state of the other.

2.4 Properties of Turing Machines

This section presents some properties of Turing Machines useful for this work.

Define the *tape equivalence* relation $\simeq$ as the binary relation that denotes an equivalence between two tapes $\mathcal{P}_1 \simeq \mathcal{P}_2$. Note that tape equivalence may be defined in various ways, consider the following two examples. The first would be simple equality when $\simeq$ is defined to be $=$, i.e., $\mathcal{P}_1 \simeq \mathcal{P}_2$ if $\forall i \in \mathbb{N} \,.\, i\#\mathcal{P}_1 = i\#\mathcal{P}_2$. The second might be equality for the first k symbols, e.g., $\mathcal{P}_1 \simeq \mathcal{P}_2$ if $\forall i \in \{0, 1, 2, \ldots k-1\} \,.\, i\#\mathcal{P}_1 = i\#\mathcal{P}_2$.

Exploiting a definition of tape equivalence, then *functional equivalence* $=_F$ can be defined for two TMs.

Definition 2.5 (Functional Equivalence). Given two TMs $\mathcal{T}_1$ and $\mathcal{T}_2$ and a tape equivalence relation $\simeq$. Then $\mathcal{T}_1 =_F \mathcal{T}_2$ the following hold:

(1) $\mathcal{T}_1 \mapsto\!\!\!\!\!\!\mapsto \mathcal{T}_1'(s_{FF}, \mathcal{P}_1')$ where $\mathcal{T}_1'(s_{FF}, \mathcal{P}_1')$ is halted iff $\mathcal{T}_2 \mapsto\!\!\!\!\!\!\mapsto \mathcal{T}_2'(s_{GF}, \mathcal{P}_2')$ where $\mathcal{T}_2'(s_{GF}, \mathcal{P}_2')$ is halted and $\mathcal{P}_1' \simeq \mathcal{P}_2'$,
(2) $\mathcal{T}_1 \perp$ iff $\mathcal{T}_2 \perp$.

That is, two TMs $\mathcal{T}_1$ and $\mathcal{T}_2$ are functionally equivalent under $\simeq$ if either: both terminate with equivalent tapes $\mathcal{P}'_1 \simeq \mathcal{P}'_2$, or both fail to terminate.

The above definition of equivalence can be extended to abstract away the tape as an input.

Definition 2.6 (Input Equivalence). Given two TMs $\mathcal{T}_1(s_1, \cdot)$ and $\mathcal{T}_2(s_2, \cdot)$ and a tape equivalence relation $\simeq$, then for all tapes $\mathcal{P}$ it holds that $\mathcal{T}_1(s_1, \mathcal{P}) =_F \mathcal{T}_2(s_2, \mathcal{P})$ under $\simeq$.

Observe that the above definition for TMs can be easily extended to UTMs and PTMs.

3 MODEL

This section formalises how fault injection can be modeled by a Parallel Turing Machine. This allows the representation and reasoning about a wide variety of fault models in a consistent framework that can support formal results regarding fault injection. To formalise fault injection involves two aspects. The first is how to formalise that notion of a "fault", or behaviour that is not part of the system/TM being considered. The second part is how to formalise the capabilities of the fault, or the fault model.

3.1 Formalising Faults

Formally a *fault* is a change of the state of the system that is not an outcome of the normal operation of the system. For the purposes of this work the source of this change is unimportant, whether it is an EMP, laser, radiation, row-hammer or other mechanism. The goal here is to precisely capture that such behaviour is *not* within this definition of the correct system behaviour and model. Note that discussion of fault models on real systems and how they relate to this work is covered in Section 7.

When considering the system as represented by a TM, faults could include: changing the states $\mathcal{S}$, the alphabet $\mathcal{A}$, the transition relation $\rightarrow$, the current state s_c, or the tape $\mathcal{P}$. In particular this concerns any change (or combination of changes) that is not performed by the normal reduction of the TM. Note that both in reality and here a "fault" could exactly achieve the results of normal operations, however this is ignored for now as this is uninteresting.

Changes to the state of a TM can effectively be considered to be adding or removing states (or both). However, observe that in both cases the significance of the states is in their existence in the transition relation. That is, if a state does not appear anywhere in the transition relation then this state can merely be considered as an additional final state that cannot be reached (except through faulting the transition relation). If a state is removed from the set of states but still appears in the transition relation then the TM becomes incorrectly defined, but will still operate as expected.

Similarly, changes to the alphabet include the adding or removing of symbols. Again observe that the significance of the addition or removal of symbols is mostly related to the transition relation rather than the definition of the alphabet itself. If an additional symbol is added that does not appear anywhere in the transition relation then (unless this symbol appears on the tape) this will be ignored by the TM. If a symbol is removed from the alphabet but still appears in the transition relation then (again) the TM becomes incorrectly defined, but will still operate as expected. Aside, in practice on some systems the adding or removing of symbols is effectively impossible. Consider a 64-bit system and the memory as arranged as words of 64-bits. Any 64-bit value is a valid symbol and so the alphabet is all 64-bit values. In practice one cannot introduce a new 64-bit value or erase one from (practical) existence, so the alphabet may be impossible to change in practice.

Changes to the transition relation are more wide ranging in their impact. These can include changing states, symbols, movements, or adding/removing entire tuples. Generally such changes affect the behaviour of the TM. Observe that without a clear limitation on the scope of changes to the transition relation the effect can range from none, to a completely new TM with new behaviour. Thus, all possible such changes are not explored here, although the rest of this paragraph identifies some smaller examples. Changing a single symbol or state in one tuple of the transition relation has a minor impact on the behaviour if that tuple is part of the (future) reductions of the TM. This can include introducing non-deterministic behaviour in an otherwise deterministic TM, or also cause the TM to halt unexpectedly. Less significant impacts can be changing a part of the computation or results. When entire tuples are added or removed this can change the behaviour in a similar manner as to changes within a tuple. Further, observe that changes to the transition relation can also include referencing states or symbols that do not exist, and this lead to the same impacts as for the changes mentioned above.

Changes to the current state are relatively straightforward. These yield a new choice from the transition relation when considering the next reduction. Although this can have significant impact on the result of the TM (or it's behaviour), the reasoning to determine the impact is minimal.

Lastly, changes to the tape are fairly wide ranging. The addition of new symbols is already partly covered by changes above. Otherwise, changes include altering the symbols on the tape, or the current position. Generally these can have significant impact on the outcome of the TM. These are particularly significant when considering a UTM since all of the other faults above are now possible merely by faulting the tape, this is exploited in the results.

Formally, a *fault* $\mathcal{T} \rightsquigarrow \mathcal{T}'$ is a transition $\rightsquigarrow$ from a TM $\mathcal{T}$ to another TM $\mathcal{T}'$. Observe that this places no particular constraints upon the similarities or differences between $\mathcal{T}$ and $\mathcal{T}'$. This is instead handled by the fault model.

It is important to remember that when considering a UTM that is simulating another TM, it is possible to produce any of the above faults (i.e. states, alphabet, transitions, state, or tape) using only a fault of the tape itself. This is significant in later results and analogous to modern systems where the program itself is loaded onto the hardware machine for execution.

3.2 Fault Models

Generally when considering fault injection, the kind of fault that can occur is limited to a specific *fault model* [10, 11]. The fault model describes how the system under consideration is altered by the fault injection. Here such a model describes the changes to the TM made by the fault, including any of the possible faults described in the previous section.

Since faults can cover many possible changes, the easiest way to capture the behaviour of the fault is to define the fault model

to operate upon the tape of a TM. This would severely limit the capabilities of the fault model to impact the behaviour of the TM. However, this can be resolved by taking the fault model to only affect the tape and then apply this to a UTM. Now the fault model has access to the tape that in turn represents all of the: state, alphabet, transition relation, current state, and tape of the TM that is being simulated.

Aside. Note that for some results later it is sufficient to reason only about faults on the tape of the TM that performs the computation. In particular the use of the UTM is only required for some fault models and some results.

Now that the injection of a fault is the alteration of the tape of a (Universal) TM, the next step is to define how such an alteration can occur in practice. Here the use of a Parallel TM provides an elegant and straightforward solution since this allows both the (Universal) TM in question and the fault model (represented by another TM) to operate on the same tape.

To represent a fault model $\mathcal{F}$ upon a TM $\mathcal{T}$ the most general way to represent this is to consider the Parallel Turing Machine $F \oplus \overline{\mathcal{T}}$ where F is the TM that implements $\mathcal{F}$ on $\overline{\mathcal{T}}$. Thus, the Parallel Turing Machine $F \oplus \overline{\mathcal{T}}$ may at any time inject the fault represented by the fault model $\mathcal{F}$ while the UTM is performing the operations of the TM $\mathcal{T}$. Note that F may have any number of transitions to induce the fault and that these may (or may not) be considered to happen during or in between operations of the UTM $\overline{\mathcal{T}}$ or the TM being simulated $\mathcal{T}$. For simplicity here the operations of the fault $\mathcal{F}$ will be assumed to occur atomically, although no result relies upon this choice.

Aside. Note that when the fault model only changes the tape of a TM $\mathcal{T}$ then the fault model upon $\mathcal{T}$ can be represented by the Parallel Turing Machine $F \oplus \mathcal{T}$ (with F adapted to implement $\mathcal{F}$ adapted appropriately).

4 COUNTERMEASURES

This section considers countermeasures as a gateway to reasoning about formal properties of fault injection. This begins by formalising what a countermeasure is, and presenting results related to countermeasures.

One of the main goals of a countermeasure is to be able to prevent a fault injection from causing "bad" behaviour or outcome [2, 3, 7, 17, 21, 26]. In most related works a countermeasure is considered to be a mechanism that prevents some particular outcome, typically defined very locally and with very specific and tight restrictions if defined at all [7, 21, 26]. Here the goal is to address countermeasures in a more general sense, albeit still related to the fault model being considered.

A countermeasure for a fault model is a mechanism that ensures the fault model cannot change the result of the computation.

Definition 4.1 (Effective Countermeasure). Given a Turing Machine $\mathcal{T}$ and fault model $\mathcal{F}$ an *effective countermeasure* $C_{\mathcal{F}}(\mathcal{T})$ for the fault model $\mathcal{F}$ ensures that $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T})}$ is functionally equivalent to $\overline{\mathcal{T}}$.

That is, given a TM $\mathcal{T}$ and a fault model $\mathcal{F}$ an effective countermeasure $C_{\mathcal{F}}(\mathcal{T})$ for the same fault model $\mathcal{F}$ ensures that the Parallel Turing Machine $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T})}$ is functionally equivalent to $\overline{\mathcal{T}}$ for all possible operations. In practice this means that in the presence of the fault model $\mathcal{F}$ implemented by the TM F operating on the effective countermeasure $C_{\mathcal{F}}(\mathcal{T})$ of the simulation of $\mathcal{T}$ no change to the simulation $\overline{\mathcal{T}}$ of $\mathcal{T}$ by the UTM can change the behaviour. Observe that functional equivalence is parametrised by a tape equivalence relation, for this section this relation is defined to be equality for the most general results.

The intuition behind the above definition is that the effective countermeasure ensures that the TM still produces the same behaviour regardless of the operation of the fault model. The lifting to simulation by the UTM is here done to ensure that the fault model can operate on any part of the TM considered (i.e., also on the state, alphabet, transition relation, and current state, not only on the tape). However, if the fault model is limited to the tape, the following (more direct) definition of an effective countermeasure can be used.

Definition 4.2 (Direct Effective Countermeasure). Given a Turing Machine $\mathcal{T}$ and fault model $\mathcal{F}$ that operates on the tape of $\mathcal{T}$, a *direct effective countermeasure* $C_{\mathcal{F}}(\mathcal{T})$ for $\mathcal{F}$ ensures that $F \oplus C_{\mathcal{F}}(\mathcal{T})$ is functionally equivalent to $\mathcal{T}$.

This definition of a direct effective countermeasure is useful when the fault model operates only on the tape, but is much more limited in fault model capabilities. The introduction of this definition here is for illustration and later results.

Proposition 4.3. *Any direct effective countermeasure is also an effective countermeasure.*

Proof. The proof is straightforward by observing that if the relation that $F \oplus C_{\mathcal{F}}(\mathcal{T})$ is equivalent to $\mathcal{T}$ holds, then by definition of the encoding to a UTM it follows that $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T})}$ is equivalent to $\overline{\mathcal{T}}$. The only delicate point is that a UTM may turn a single transition of $\mathcal{T} \mapsto \mathcal{T}'$ into many transitions. However, since the translation of the fault model $\mathcal{F}$ has been adapted to preserve the effect on (the encoded) tape, this has no effect on the result. □

For effective countermeasures the choice here to parametrise by the fault model is to allow finer grained reasoning about the fault and effectiveness of the countermeasure. This is due to infeasibility of general countermeasures to all possible faults, as intuited by the following result.

Theorem 4.4. *It is impossible to have an effective countermeasure against all fault models.*

Proof. The proof is by contradiction. Given an arbitrary TM $\mathcal{T}$ consider whether or not this TM terminates.

- If $\mathcal{T} = \mathcal{T}(s, \mathcal{P})$ terminates by reducing to a TM $\mathcal{T}(s', \mathcal{P}')$ then consider the fault model $\mathcal{F}$ that permutes the symbol to the right of the symbol at the current position of the tape and then terminates. That is, $\mathcal{F}$ indexes the alphabet $\mathcal{A}$ by the natural numbers and changes the symbol a_i at $pos(\mathcal{P}')+1$ to be a_j where $j = (i+1)\%|\mathcal{A}|$ where % is the modulo operator, and then terminates.
  Now consider $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s, \mathcal{P}))}$ where $C_{\mathcal{F}}(\mathcal{T}(s, \mathcal{P}))$ is an effective countermeasure. Since $C_{\mathcal{F}}(\mathcal{T}(s, \mathcal{P}))$ is an effective

countermeasure, it follows that there exists a sequence of reductions $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))} \rightarrowtriangle F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s',\mathcal{P}'))}$ such that $\overline{C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))}$ has terminated (with the same tape $\mathcal{P}'$ as $\mathcal{T}'\mathcal{P}'$ at termination). Observe that so far the fault TM F has not reduced.
Now since $\overline{C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))}$ has terminated, the only possible reduction of $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s',\mathcal{P}'))}$ is $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s',\mathcal{P}'))} \rightsquigarrow F' \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s',\mathcal{P}''))}$ from $\mathcal{F}$ and by definition of $\mathcal{F}$ it must be that $\mathcal{P}' \neq \mathcal{P}''$. (Aside, note that since the fault modifies the symbol at a position different to the current position the only way for the countermeasure to check this position is to move the current position, which in turn means that $\overline{C_{\mathcal{F}}(\mathcal{T}(s',\mathcal{P}'))}$ has not terminated, yielding contradiction.) However, this contradicts the definition of an effective countermeasure.

- If $\mathcal{T}(s,\mathcal{P})$ does not terminate then consider the fault model $\mathcal{F}$ that given a TM $\mathcal{T} = (\mathcal{S}, \mathcal{A}, \rightarrow, s_c, \mathcal{P})$ changes the current state to some s_F in the final states of $\mathcal{T}$. This immediately puts the TM into a final and terminated state.
  Now observe that since $\mathcal{T}(s,\mathcal{P})$ never terminates $\overline{\mathcal{T}(s,\mathcal{P})}$ never terminates. It follows by definition of an effective countermeasure that $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))}$ must also never terminate. Now consider the PTM after the reduction(s) $F \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))} \rightsquigarrow F' \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$. Since $\mathcal{F}'$ has terminated, the only possible way for another reduction to occur is via a reduction of $\overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$, however the fault has put $\mathcal{T}(s_F,\mathcal{P})$ into a terminated state. Now there are two possible scenarios.
  (1) If $\overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$ is terminated then this contradicts the definition of an effective countermeasure since $F' \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$ is terminated but $F' \oplus \overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$ must also not terminate.
  (2) If $\overline{C_{\mathcal{F}}(\mathcal{T}(s_F,\mathcal{P}))}$ has a reduction then this reduction must be due to some part of the countermeasure $C_{\mathcal{F}}(\cdot)$ since $\mathcal{T}(s_F,\mathcal{P})$ and the UTM simulation. If these reductions are only due to the UTM simulation then $C_{\mathcal{F}}(\cdot)$ since $\mathcal{T}(s_F,\mathcal{P})$ will terminate and yield contradiction.
  Thus, the only possible outcome is that the countermeasure continues to reduce infinitely. Now consider two possibilities:
    (a) If the reduction is due solely to the countermeasure itself and so $C_{\mathcal{F}}(\cdot)$ reduces regardless of what TM is introduced into $\cdot$ then this contradicts termination with a TM that terminates. Thus again yielding contradiction.
    (b) The only other possibility is that the countermeasure $C_{\mathcal{F}}(\cdot)$ can detect whether an arbitrary TM $\mathcal{T}$ given to it will halt or not, which is known to be contradictory (i.e. this would solve the halting problem). □

The intuition behind this proof is as follows. If the machine terminates and produces output, then a fault that changes the output effectively changes the program behaviour. The only way for a program to check the output at all times is to not terminate. If the program never terminates, a fault can put the (simulated) program into a terminated state (e.g., jump past last instruction). Therefore, any non-termination that cannot be in the (simulated) program must be in the UTM (e.g., operating system). However, now any program must fail to terminate, so this cannot be an effective countermeasure for trivial programs like "always return 1".

Corollary 4.5. *It is impossible to have a direct effective countermeasure against all fault models.*

Proof. The result is achieved in a similar manner to Theorem 4.4 by considering a TM $\mathcal{T} = \mathcal{T}(s,\mathcal{P})$ and tape $\mathcal{P}$.

- If $\mathcal{T}(s,\mathcal{P})$ terminates at $\mathcal{T}(s',\mathcal{P}')$ then consider when If $C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))$ terminates at $C\mathcal{F}\mathcal{T}(s',\mathcal{P}')$ and use the fault model that changes the symbol to the right of the current position as before.
- If $\mathcal{T}(s,\mathcal{P})$ does not terminate, then use a fault model that inserts a symbol a_X (that does not appear in any transition relation tuple of $C_{\mathcal{F}}(\mathcal{T}(s,\mathcal{P}))$) onto the tape at the current position. Since there is no transition that applies for the symbol a_X then the TM will halt. □

The above results show that effective countermeasures to all possible fault models are impossible. Indeed, just within the scope of some carefully chosen fault models there is no way to protect an arbitrary TM from being affected. However, this is not a particularly unexpected result, since if one can change any aspect of a system at any time, it is (unsurprisingly) impossible to prevent some faulty behaviour.

Observe that in Corollary 4.5 the non-terminating condition requires the ability for the fault model to inject a symbol a_X onto the tape that is not in the alphabet of the original TM. This may appear unsatisfactory in its triviality since creating a new symbol for the TM is a simple way to break the transition relation, and indeed unreasonable since one can consider implementation on a real system where all valid words can be enumerated and so no new symbols created. This is a limitation of a fault model that can only modify the tape and so have very limited access to the TM they fault. Without allowing new symbols it is trivial to design a non-terminating TM that would be impossible to fault into termination by modifying the tape, simply by having two states and for all symbols on the tape simply alternate moving left and right (transitioning into the other state). However, such a TM is not very interesting as an actual program.

The above considered, it is more interesting is to consider how such approaches can be employed to formalise the effectiveness of countermeasures against specific fault models, or within specific contexts, and while allowing arbitrary TMs that can do interesting computations.

5 EXAMPLE COUNTERMEASURE

Consider the fault model $\mathcal{F}$ from the first case of Theorem 4.4; i.e., the fault model that changes a single symbol on the tape. An effective countermeasure can be constructed by altering structure of the tape to have triples to represent each symbol, i.e., the tape $a_0, a_1, a_2, \ldots$ would be represented as $a_0, a_0, a_0, a_1, a_1, a_1, a_2, a_2, a_2, \ldots$. The TM can then be adapted to instead take the majority of the symbols as the symbol read, thus preventing any single symbol modification from changing the behaviour.

This kind of countermeasure is designed to be effective against very limited fault injection attacks and has parallels in various real

$(s_w, a_0, a_0, +1, s_{w0})$
$(s_w, a_1, a_1, +1, s_{w1})$
$(s_{w0}, a_0, a_0, -1, s_{iiix})$
$(s_{w0}, a_1, a_1, +1, s_{w01})$
$(s_{w1}, a_0, a_0, +1, s_{w01})$
$(s_{w1}, a_1, a_1, -1, s_{jjjy})$
$(s_{w01}, a_0, a_0, -2, s_{iiix})$
$(s_{w01}, a_1, a_1, -2, s_{jjjy})$
$(s_{iiix}, a_0, a_i, +1, s_{iix})$
$(s_{iiix}, a_1, a_i, +1, s_{iix})$
$(s_{iix}, a_0, a_i, +1, s_{ix})$
$(s_{iix}, a_1, a_i, +1, s_{ix})$
$(s_{ix}, a_0, a_i, m'_1, s_x)$
$(s_{ix}, a_1, a_i, m'_1, s_x)$
$(s_{jjjy}, a_0, a_i, +1, s_{jjy})$
$(s_{jjjy}, a_1, a_i, +1, s_{jjy})$
$(s_{jjy}, a_0, a_i, +1, s_{jy})$
$(s_{jjy}, a_1, a_i, +1, s_{jy})$
$(s_{jy}, a_0, a_i, m'_2, s_j)$
$(s_{jy}, a_1, a_i, m'_2, s_j)$.

Figure 1: A new transition relation for a simple TM that uses majority of three consensus to be an effective countermeasure against single symbol fault injection.

world applications. Examples of this approach used in practice are: Redundant Arrays of Independent Drives (RAID)s, Error-Correcting Codes (ECC) in network communication, & consensus multi-sensor systems in aviation.

The following shows how this can be implemented for a single state s_w. Assume that $\mathcal{A} = \{a_0, a_1\}$ for simplicity. The transitions $(s_w, a_0, a_i, m_1, s_x)$ and $(s_w, a_0, a_j, m_2, s_y)$ can be represented by the transition relation in Fig. 1. The first state s_w now proceeds to the state s_{wk} when the first symbol is a_k. The states s_{w0} and s_{w1} and s_{w01} are used to determine how many instances of each symbol (a_0 or a_1) have been observed so far, proceeding to one another as required. Once a majority (two of the same symbol) have been detected, the movement returns to the initial position when at state s_w and writes out three copies of either a_i or a_j as required. After writing the third symbol, the movement is then either m'_1 or m'_2 that has been adapted from m_1 and m_2 (respectively) to account for the tripling of symbols on the tape.

To formalise an effective countermeasure now here define the *triple-equivalence* $\mathcal{P}_1 =_3 \mathcal{P}_2$ of two tapes $\mathcal{P}_1$ and $\mathcal{P}_2$ to be true if for all $i \in \{0, \ldots, \infty\}$ then at least two of: $(3i)\#\mathcal{P}_1 = (3i)\#\mathcal{P}_2$ and $(3i+1)\#\mathcal{P}_1 = (3i+1)\#\mathcal{P}_2$ and $(3i+2)\#\mathcal{P}_1 = (3i+2)\#\mathcal{P}_2$ hold. Observe that $\mathcal{P}_1 =_3 \mathcal{P}_2$ holds when for all triples of symbols, the two tapes disagree on at most one symbol.

Theorem 5.1. *Triplicating the symbols on the tape and adapting the TM appropriately is a direct effective countermeasure against a single symbol alteration fault model.*

Proof. If the fault occurs after the termination of the TM then it is trivial to show that $\mathcal{P} =_3 \mathcal{P}'$ where $\mathcal{P}'$ is produced by $\mathcal{F}$. The alteration of the tape also does not affect the state of the terminated TM and so cannot introduce new reductions.

If the fault occurs during the reductions of the TM $C_{\mathcal{F}}(\mathcal{T})$ then it is straightforward (if tedious) to show that each reduction of $\mathcal{T}$ is now mapped to several reductions of $C_{\mathcal{F}}(\mathcal{T})$ but that the majority will win out if the fault has changed one symbol. (Aside, if the fault occurs on part of the tape where no further reads by the TM are done, then this trivially reverts to the previous case.) □

The only delicate point in the above proof is to recall that the tape must start in a correctly configured state. Note that if the tape begins with one triple of symbols already being mismatched (e.g. a_0, a_1, a_0) then a single change can reverse the majority.

One may observe that the triple-equality relation is much more flexible than is required for a single fault to a single symbol. Indeed, if the fault model is allowed to fault a single symbol out of each three consecutive symbols on the tape, then the same approach is still an effective countermeasure.

Corollary 5.2. *Triplicating the symbols on the tape and adapting the TM appropriately is a direct effective countermeasure against a fault model that alters at most a single symbol for any three symbols indexed $a_{3i}, a_{3i+1}, a_{3i+2}$ for $i \in \{0, \ldots, \infty\}$.*

Proof. This is a straightforward adaptation of the proof of Theroem 5.1. □

These results show that although the general results of Section 4 may appear daunting and discourage the investigation of effective countermeasures, in practice effective countermeasures can exist and indeed some existing approaches are effective countermeasures to some fault models.

6 VARIATIONS

This section discusses some natural extensions to the models and results above that can be useful. Since the goal of the above results and examples is to demonstrate that formalising fault injection can be done and the limits of such formal approaches, this section explores two ways to extend these limits using similar methods.

6.1 Probabilistic Turing Machines

The results here have exploited simple TMs in the sense that they have been assumed to have deterministic or non-deterministic behaviours. There is also the possibility to reason over probabilistic Turing Machines where the transition relation also considers the probability when multiple possible transitions can be taken. For example, when the transition relation includes both $(s_{w1}, a_{x1}, a_{y1}, m_1, s_{z1})$ and $(s_{w2}, a_{x2}, a_{y2}, m_2, s_{z2})$ where $s_{w1} = s_{w2}$ and $a_{x1} = a_{x2}$ (where the rest of a_{y1}, a_{y2} and m_1, m_2 and s_{z1}, s_{z2} may or may not be equal). Instead of a non-deterministic choice between them, the transition relation could be augmented with a probability distribution over all transitions that share the same s_w and a_x (i.e., of the form $(s_w, a_x, \cdot, \cdot)$).

This would be particularly useful for quantifying the probability of faults or fault injection attacks occurring. In practice a very

low probability could be given to a rare fault injection, and then probabilistic results about the countermeasure used.

For example, consider when the fault may be a memory corruption due to background radiation. This could be formalised by defining a fault model that randomly changes the value of one symbol on the tape (e.g., flips a bit to create a new symbol). In practice this could be quantified with a probability such that the fault happens with only ρ probability for any given tick of the system. Now, the fault could be represented by F_ρ such that when $F_\rho \oplus \mathcal{T}$ operates at any given tick the possible reductions are:

- $F_\rho \oplus \mathcal{T} \rightsquigarrow F'_\rho \oplus \mathcal{T}$ with probability ρ (where F'_ρ and $\mathcal{T}'$ indicate the fault injection has occurred) and
- $F_\rho \oplus \mathcal{T} \mapsto F_\rho \oplus \mathcal{T}'$ with probability $1-\rho$ (where $\mathcal{T} \mapsto \mathcal{T}'$).

Note that there is no limitation here that F'_ρ does not cause further faults, and indeed in the case of modeling atmospheric radiation it may be best to define $F'_\rho = F_\rho$ and thus allow any number of faults.

Now it would be possible to formalise that given a TM $\mathcal{T}$ that terminates in k reductions, then given $F_\rho \oplus \mathcal{T}$ would have had a fault injection occur with probability $(1-\rho)^k$ after k ticks when $\mathcal{T}$ would be expected to terminate.

This gives an immediate probability of a fault, but does not indicate whether this causes a significant problem (since the fault injection may or may not have changed the behaviour). However, assuming that a fault has occurred, this can provide a probability of a fault injection damaging the safety or security of the program.

Now consider the same fault model and the countermeasure of triplicating the tape as in Section 5. To obtain a useful result it is helpful (but not necessary, see the end of this section) to assume that the tape is finite. Thus, assume that the tape is has length l and that the probability of the fault injection happening to each symbol is uniform. Now, to cause a fault injection to two symbols in the same triple can be calculated to be $\rho \times \rho \times \frac{3}{l} \times \frac{2}{3} \times \frac{1}{|\mathcal{A}|}$ where the fault must occur twice $\rho \times \rho$ and must be in the same triple $\frac{3}{l}$ and must affect different symbols $\frac{2}{3}$ and must also map them both to the same symbol $\frac{1}{|\mathcal{A}|}$. (In practice the probability is even lower since this does not consider faulting a symbol to the same as the original, or the two faults occurring one before and one after the first faulted symbol is read and over-written.)

Observe that in such a scenario even though the countermeasure can be proven not to be effective, if one is willing to accept a very small probability of a fault then the countermeasure may be acceptable.

Note that above the tape was assumed finite since this simplifies the reasoning and is more closely related to real world systems. However, the only place this is used is in the probability of the fault occurring (the value $\frac{3}{l}$). The same equations hold if one assumes that faults are limited to the first l symbols of the tape, and are uniformly distributed within these symbols. Indeed, one can still produce results on TMs with infinite tapes by merely limiting the fault model.

Definition 6.1 (ϵ-Effective Countermeasure). Given a probabilistic Turing Machine $\mathcal{T}_\pi$ and a probabilistic fault model $\mathcal{F}_\rho$ and a value ϵ an *ϵ-effective countermeasure* $C_{\mathcal{F}_\rho}(\mathcal{T}_\pi)$ for the fault model $\mathcal{F}_\rho$ ensures that $F_\rho \oplus \overline{C_{\mathcal{F}_\rho}(\mathcal{T}_\pi)}$ is probabilistically functionally equivalent to $\overline{\mathcal{T}_\pi}$ with probability $> 1-\epsilon$.

The above definition is a straightforward adaptation where the TM and fault model are both now represented by probabilistic TMs. The other extension is to define the functional equivalence relation to account for probabilistic transitions of a TM (this is non-trivial in the general sense, but results here rely only upon the existence of this relation).

Theorem 6.2. *Given a probabilistic Turing machine $\mathcal{T}_\pi$ with alphabet $\mathcal{A}$ that terminates within k ticks and a probabilistic fault model $\mathcal{F}_\rho$ (that changes one symbol uniformly at random, and within the first l symbols of the tape with uniform probability ρ then transitions to its initial state), Then there exists an ϵ-effective countermeasure for $\epsilon < \rho \times \sum_{i=1\ldots k-1} Pr(i; k-1, \rho) \times \frac{3i}{l} \times \frac{2}{3} \times \frac{1}{|\mathcal{A}|}$.*

Proof Sketch. The proof is by triplicating the tape as in Theorem 5.1 and then calculating the probability of two faults within the same triple and mapping to the same symbol within the number of ticks required for $\mathcal{T}_\pi$ to terminate. This can be approximated by

$$\rho \times \sum_{i=1\ldots k-1} \Pr(i; k-1, \rho) \times \frac{3i}{l} \times \frac{2}{3} \times \frac{1}{|\mathcal{A}|} \ .$$

This is the probability of a fault occurring ρ multiplied by the sum of the probabilities of i further faults occurring $\Pr(i; k-1, \rho)$ (where $\Pr(i; k-1, \rho)$ is the standard probability mass function for a binomial distribution) and one of these other faults being in the same part of the tape as a prior fault $\frac{3i}{l}$ and modifying different parts of the same triple $\frac{2}{3}$ and both changing to the same symbol $\frac{1}{|\mathcal{A}|}$. This is sufficient to prove when $\mathcal{T}_\pi$ is deterministic and using Definition 2.5, when $\mathcal{T}_\pi$ has probabilistic branches in execution then an appropriate probabilistic functional equivalence relation must be used. □

Note that the above result is a sketch to focus on the concepts and can be made more precise (but less clear). In particular this is an over approximation since it does not consider for example when: faults occur and are fixed by the operation of $\mathcal{T}_\pi$, faults change a symbol to itself, $\mathcal{T}_\pi$ terminates in less than k ticks, and faults over-write other faults.

On the other hand, the above illustrates that the approach of triplicating the tape as in Theorem 5.1 is likely to be faulted if the $\mathcal{T}$ does not terminate. This could in turn be countered by having $\mathcal{T}$ inspect/repair the entire tape (of length l) before faults could (probabilistically) build up. However, this becomes non-trivial to design and reason about.

The above illustrates how one can construct an effective countermeasure up to a confidence level ϵ when an absolute countermeasure may be impossible or too expensive.

6.2 Time

The above extension for probability already exploits some notion of a "tick" or clock cycle that implies time for some clarity. However, time can also be made an explicit part of the model in a more concrete manner that allows use of time to understand and control

fault injections. For example, a fault could only occur before or after a certain amount of time and thus proofs could be constructed to show that a system could recover from a fault or have an effective countermeasure if the fault occurs only during certain times.

Observe that incorporating time into the model brings the model closer to real world systems even if time is handled relatively abstractly. As in Section 6.1 a simple notion of a tick or clock cycle for the reductions being applied to the normal behaviour of the system would allow reasoning about the computation time and thus the scope of faults. Further, by defining the fault reductions $\rightsquigarrow$ to take zero time this models the atomicity of a fault without the assumption that faults are atomic (as generally assumed here, although not required for any result).

In addition to including a notion of time, this allows the modeling if faults that occur only rarely or allows the definition of effective countermeasures that can defend a (finite) system against recurring but rare faults.

Theorem 6.3. *Given a Turing Machine $\mathcal{T}$ that operates within a finite tape of length l and a fault model $\mathcal{F}$ that changes exactly one symbol of the tape and then has no affect for greater than $6l + 8$ ticks, there exists an effective countermeasure.*

Proof Sketch. The countermeasure is to triplicate the tape as in Theorem 5.1 and modify $\mathcal{T}$ to some $\mathcal{T}'$ that takes into account the triplicating of the tape as described in Section 5. Now further modify $\mathcal{T}'$ to created $\mathcal{T}^R$ that after each transition of $\mathcal{T}$ into a state s_x of $\mathcal{T}$ to go into a "check state" s_{cx}. Now this check state s_{cx} initialises a traversal of the entire tape length that repairs any mismatch of symbols before ending in the original position and in the state equivalent to c_x of $\mathcal{T}$ in the modified $\mathcal{T}^R$. Observe that to traverse (and if necessary repair) the tape takes $2 \times l \times 3 + 2$ transitions ($2 \times l$ to traverse the length l of the tape and return to the original position, and $\times 3$ for the triplicating of the tape, and +2 to go back and rewrite a symbol if necessary). Further, the modified transitions of $\mathcal{T}$ in $\mathcal{T}^R$ can be achieved with a maximum of 6 transitions. Thus, if faults occur greater than $6l + 8$ ticks apart, then this countermeasure is effective. □

Observe that the above countermeasure is very expensive in time taken as the entire tape is checked for faults after every transition of the original TM (and also that no optimisations are assumed in traversing and repairing the tape). In practice this would be equivalent to checking parity and repairing the entire system memory on every CPU instruction, so far too expensive in practice. However, combined with probabilistic tolerances and likelihood of faults (see Section 6.1) this could provide a good assurance of reliability and security of a system with regular checks for memory corruption. (Note that this is conceptually very similar to the current systems used by modern hard disks to routinely check for bad sectors and repair/replace them.)

7 DISCUSSION

This section briefly discussed the results and variations, and a broader understanding of their connections and implications.

7.1 Finite Turing Machines

Observe that in Section 6 when the scope of some computation or effects are limited to a finite tape it is possible to prove interesting results that are less clear on an infinite tape.

The results here have all considered the spirit of the original definition of Turing Machines where the tape is infinite. However, in practice many modern systems have many effectively finite components (finite memory, finite storage, finite alphabet, finite states) and so considering a finite tape could be a reasonable limitation. The results of Section 4 exploit the infinite nature of TMs (as this is used in some proofs of the halting problem) and so the results must be redone for a finite system. However, the key concepts can be applied regardless of the finite or infinite nature of the system, as the key emphasis of this work is to encourage a formal proof of effectiveness for a countermeasure, not to discourage such efforts.

7.2 Fault Models

There are many different fault models described in the literature that define how a fault can affect a system [9, 27]. The focus in this work has been on formalising fault models, and doing this by exploiting the properties of TMs and UTMs as a basis for computing. Since modern systems are mostly derived from von Neumann architectures that are close to TMs in practice, this applies naturally. The core point to recognise is that by considering a UTM where a program is written on the tape and interpreted by the Universal Turing Machine, this very closely matches how current hardware interprets programs stored in some form of memory.

Focusing on the different kinds of fault models, most can be considered as modifications of the memory rather than defined separately. Fault models such as *control flow modification* [9, 23], *instruction faults* [10], *non-operations* [9, 21], *value modification* [9, 30] and *bit-flips* [9, 27] amongst others can all be captures by the modifications here. Consider the following brief examples.

Control flow modification can be modeled here in different ways. A conditional control flow can be modified by changing the symbol for the value that is used for the condition. Alternatively, the target address of a jump can be modified by changing the value of the address in the memory (the tape). Another method would be to change the symbol for the transition relation to change the next state in the TM on the tape of the UTM and so change control flow that way.

Instruction faults are straightforward by changing the symbol that represents the instruction on the tape of the UTM to a different symbol. This is most trivial when the tape exactly maps to memory and thus each symbol of the tape is an instruction. (Aside: of course this is easiest to handle on systems with fixed instruction size, but even with variable instruction size the change of a single hardware word can be considered equivalent even if multiple instructions can potentially be effected by one symbol. Vice versa is the symbols are at the level of bytes then the fault may need to change multiple symbols, but is conceptually straightforward.)

Non-operations or *instruction skips* can be defined to be modification of the instruction (see above) to be one that has no effect on the system (and NOP in the UTM/hardware), or by faulting the transition related stored on the tape of the UTM to skip over the next instruction.

Value modification can be done directly to a TM without the need of a UTM when the values are stored on the tape, merely change the appropriate symbols on the tape.

Bit-flips as essentially a special instance of other fault models where only some modifications are allowed and thus they can be modeled as above. (Aside: such hardware/cause focused fault models are interesting to consider in some domains, but their representation here is a limited instance of a more general fault model.)

Observe that this allows many common and widely considered fault models to be easily represented and reasoned about using the techniques here. The main limitation of such approaches is in the complexity of reasoning about the full alphabet and instruction set for a modern system. However, this only makes the proofs and effort required larger, not significantly more complex than the examples shown here.

8 CONCLUSIONS

Fault injection can be formalised to allow reasoning on the affects of a fault, and also formalising results related to fault injection (as robustness measures or attacks) and countermeasures. This allows results that can prove (or disprove) whether a fault injection can have an effective countermeasure. Generalising this allows for various properties about different systems to be proved to be robust and secure, in particular by proving that a countermeasure is effective in the presence of fault injection attacks.

The results and examples here prove both: it is *impossible to have an effective countermeasure against arbitrary fault injection*, and that effective countermeasures can be created. The variations illustrate how to extend to probabilistic behaviours with probabilistic countermeasures, and to time to have effective rate-limited fault injection countermeasures.

Future Work. There are two clear directions of interesting future work that can be extended from this paper. The first is in the direct applications of using the methods developed here to prove the robustness and security of systems to specific kinds of faults, and the effectiveness of specific ountermeasures. The other is to consider the relations to other kinds of fault tolerance and related systems and results since these may be able to benefit each other [5, 6].

REFERENCES

[1] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2004. The Sorcerer's Apprentice Guide to Fault Attacks. *IACR Cryptology ePrint Archive* 2004 (2004), 100. http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html#Bar-ElCNTW04

[2] Guillaume Barbu, Philippe Andouard, and Christophe Giraud. 2013. Dynamic Fault Injection Countermeasure. In *Smart Card Research and Advanced Applications*, Stefan Mangard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

[3] Patrick Baudin, Jean-Christophe Filliatre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2009. ACSL: ANSI/ISO C Specification Language, version 1.4. *CEA* 6 (2009). http://frama-c.com/download/acsl_1

[4] George S Boolos, John P Burgess, and Richard C Jeffrey. 2002. *Computability and logic*. Cambridge University Press.

[5] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[6] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[7] Maria Christofi, Boutheina Chetali, and Louis Goubin. 2013. Formal verification of an implementation of CRT-RSA Vigilant's algorithm. In *PROOFS Workshop: Pre-proceedings*. 28.

[8] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, Philippe Orsatelli, Philippe Maurine, and Assia Tria. 2012. Injection of transient faults using electromagnetic pulses-Practical results on a cryptographic system-. *IACR Cryptology EPrint Archive* 2012 (2012), 123.

[9] Thomas Given-Wilson, Annelie Heuser, Nisrine Jafri, and Axel Legay. 2019. An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. *Concurr. Comput. Pract. Exp.* 31, 23 (2019). https://doi.org/10.1002/cpe.4794

[10] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. 2017. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*. IEEE, 293–300. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.250

[11] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. 2020. Combined software and hardware fault injection vulnerability detection. *Innovations in Systems and Software Engineering* (2020).

[12] Sylvain Guilley, Laurent Sauvage, Jean-Luc Danger, and Nidhal Selmane. 2010. Fault injection resilience. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE, 51–65.

[13] Armin Hemmerling. 1979. Systeme von Turing-Automaten und Zellularräume auf rahmbaren pseudomustermengen. *Elektronische Informationsverarbeitung und Kybernetik* 15, 1/2 (1979), 47–72.

[14] F. C. Hennie and R. E. Stearns. 1966. Two-Tape Simulation of Multitape Turing Machines. *J. ACM* 13, 4 (Oct. 1966), 533–546. https://doi.org/10.1145/321356.321362

[15] R. Herken. 1995. *The Universal Turing Machine A Half-Century Survey*. Springer Vienna. https://books.google.be/books?id=YafIDVd1Z68C

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News*. IEEE Press, 361–372.

[17] Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. 2013. Towards the Hardware Accelerated Defensive Virtual Machine – Type and Bound Protection. In *Smart Card Research and Advanced Applications*, Stefan Mangard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

[18] Albert Ma and Michael Zhang. 2001. Computer system architecture. *Computer* 6 (2001), L1–1.

[19] Timothy C May and Murray H Woods. 1978. A new physical mechanism for soft errors in dynamic memories. In *Reliability Physics Symposium, 1978. 16th Annual*. IEEE, 33–40.

[20] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 77–88.

[21] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. 2014. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering* 4, 3 (2014), 145–156.

[22] John von Neumann. 1945. *First Draft of a Report on the EDVAC*. Technical Report.

[23] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. 2014. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 213–222.

[24] Rui Qiao and Mark Seaborn. 2016. A new approach for rowhammer attacks. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 161–166.

[25] Peng Qu, Jin Yan, You-Hui Zhang, and Guang R. Gao. 2017. Parallel Turing Machine, a Proposal. *Journal of Computer Science and Technology* 32, 2 (01 Mar 2017), 269–285. https://doi.org/10.1007/s11390-017-1721-3

[26] P. Rauzy and S. Guilley. 2014. Countermeasures Against High-Order Fault-Injection Attacks on CRT-RSA. *arXiv e-prints* (Dec. 2014). arXiv:cs.CR/1412.0600

[27] Lionel Rivière, Julien Bringer, Thanh-Ha Le, and Hervé Chabanne. 2015. A novel simulation approach for fault injection resistance evaluation on smart cards. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 1–8.

[28] Cyril Roscian, Jean-Max Dutertre, and Assia Tria. 2013. Frontside laser fault injection on cryptosystems-Application to the AES'last round. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*. IEEE, 119–124.

[29] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[30] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining symbolic execution and model checking for data flow testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 654–665.

[31] A. M. Turing. 1936. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2, 42 (1936), 230–265.

[32] Alan M. Turing. 1937. Computability and lambda-Definability. *Journal of Symbolic Logic* 2, 4 (1937), 153–163.
[33] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. 2011. The fault attack jungle-a classification model to guide you. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE, 3–8.
[34] Gaoli Wang and Shaohui Wang. 2010. Differential fault analysis on PRESENT key schedule. In *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE, 362–366.
[35] Juraj Wiedermann. 1984. *Parallel Turing machines*. Department of Computer Science, University of Utrecht The Netherlands.
[36] Keun Soo Yim. 2016. The Rowhammer Attack Injection Methodology. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE, 1–10.