# Software Hazard Mining

Jeffrey Voas
Reliable Software Technologies
jmvoas@rstcorp.com

## Abstract

Software fault injection is a process that discovers how "badly" software can behave after its state gets corrupted. Fault injection results are sometimes viewed suspiciously since state corruption is hypothetically-based. In response to this suspicion, this paper explores the potential return-on-investment when artificial state corruptions are used. We will primarily focus on fault injection's unique ability to reveal hazards that were inadvertently overlooked during software requirements and design. To our knowledge, this application of fault injection has never been exploited.

## 1 The Problem

Mining for precious metals and diamonds are probabilistic activities. To obtain 1 gram (1/28 ounce) of diamonds it is often necessary to remove 25 tons of sand [2]. For each pea-sized quarter ounce of gold, an average of one ton of rock - about 19 cubic feet - have to be dislodged by a combination of drilling and blasting [5]. In both situations, the *noise-to-signal* ratio is quite great. But that does not stop continued exploration by companies or individuals that seek valuable natural resources.

Debugging programs for errors is analogous to mining. For reliable software programs (i.e., programs that rarely fail), debugging will often be a slow process because the fault densities are small. Faults with large densities are easier to discover.

*Hazard analysis* is the process of defining all known software hazards during the requirements phase. Often times, hazards are accidentally omitted during the process, and this can lead to serious safety deficiencies in the finished code. Kletz states [1]:

> The mistake made by many analysts is to quantify (with even greater accuracy) the particular hazards that they have thought of and and fail to foresee that there are other hazards of much more importance.

This paper will explore a process for trying to uncover new hazards that were overlooked during the requirements phase. We term this process *hazard mining*. Hazard mining deeply

probes the semantics of the code while the code executes in attempts to flush out forgotten hazards that need to be protected against.

Our approach is based on software *fault injection*. The goal of software fault injection is to observe whether a program can behave in certain predefined, undesirable manners [4]. Software fault injection is a dynamic analysis that can mine in the internals of the software and discover corrupted program states that force hazardous output events to occur. To date, this has been a key application of this software assessment technology. Interestingly, fault injection can also be used to reveal whether the software can behave in ways which previously had not been considered as undesirable. This is the goal of hazard mining.

If fault injection reveals hazardous software behaviors that were not previously considered as possible, an opportunity is afforded to take defensive measures that disallow the software from exhibiting those behaviors in the future. Thus mining inside of the "guts" of the software while it executes presents an immediate opportunity for safety enhancements if discoveries of hazardous behavior are observed. And if overlooked classes of hazards are discovered, then the safety requirements for the software can be made more complete.

Software fault injection is a "what if" analysis technique that can forcefully corrupt the information in a program's state (as well as mutate code) [6]. This paper employs the approach of corrupting program states. The reason for this is simple: one particular corruption of a specific program state for a given test execution of the software can simulate a wide class of code mutations. Further, methods for performing the instrumentation to mutate code are not nearly as sophisticated as those for corrupting internal states.

Most state corruptions employed by fault injection are termed as *artificial*, which simply means that they were not naturally caused by the program. The key questions when artificially modifying program states are:

> *(1) Do forced modifications of a program's state to a state that the program (or its external environment) did not naturally create provide any insight into how badly the program might actually behave in the future? (2) Or are artificially manufactured program state anomalies totally void of real-world implications?*

These questions together form the "reality" issue that has long plagued fault seeding, fault insertion, and fault injection techniques. Clearly, if artificially manufactured program state anomalies are totally void of real-world insights, fault seeding, fault insertion, and fault injection are useless. If artificially manufactured program state anomalies can provide valid predictions, then the question becomes: "how can we get the most information from the results?"

Before we start addressing these questions, let's first discuss in greater depth the different ingredients needed to perform fault injection. To begin, there are two unique classes of events that must be defined by the person performing fault injection. First, there are those events that will be forcefully injected into the state of the application software as it executes (this is where the "injecting" is taking place). These events are termed *data anomalies*. Examples

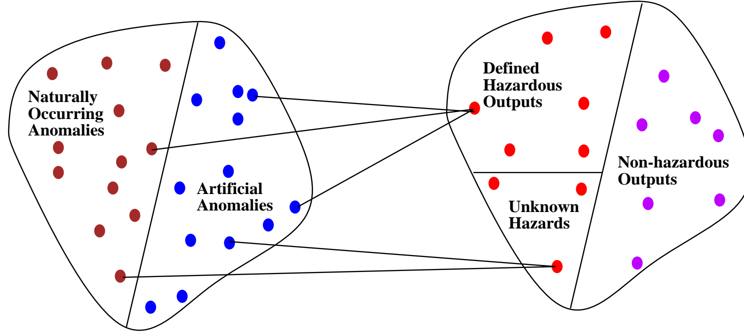


Figure 1: Natural and Artificial Anomalies Result in Hazards

here would include corrupting a pointer, modifying the value that as programmer variable currently contains, or slowing down a computation.

The second set of events are classes of functional behavior (*i.e.*, output events) that the user does not want the application software to exhibit. Examples here include hazardous output states (that are defined during hazard analysis) or calls to system-level utilities that the application software should not make. This second class of events is referred to as *output anomalies*. What constitutes as an output anomaly must be defined with respect to the state of the system that the software resides in or controls. In safety-critical systems, output anomalies are hazards.

The *reality* issue is particularly important since fault injection often creates data anomalies using *pseudo-random* number generation [7]. There are two basic ways in which pseudo-random number generation is employed to corrupt program states: (1) by changing a value to a distinct value (that is based on the original value), and (2) by changing a value to something that is totally independent of the original value.

If there is no foreseeable way in which a particular data anomaly could occur during the execution of the software in its regular environment (other than by force), then the benefit of performing fault injection may seem negligible. But interestingly enough, even though a data anomaly may not be representative of naturally occurring state corruptions, unnatural data anomalies can still forewarn about unanticipated output anomalies. Thus even inappropriate choices of data anomalies can reveal undesirable output behaviors that were not known to be possible. This suggests that fault injection can play a complimentary role to hazard and requirements analysis by testing the thoroughness of these analyses.

Figure 1 shows an example of this. In Figure 1, we see two types of anomalies: (1) naturally occurring, and (2) artificial. Three types of output events are defined: (1) non-hazardous, (2) known hazards, and (3) hazardous (but not known to be). This last category is most disturbing because the software requirements have failed to alert the software engineers to protect against such outputs. Thus these events are unlikely to be accounted for by the early design-for-safety techniques (such as fault-tree analysis).

Figure 1 illustrates possible mappings for the two classes of hazards. In one mapping,

two different artificial anomalies and one naturally occurring anomaly each cause the same defined hazardous output to occur. The other mapping illustrates an artificial anomaly and a naturally occurring anomaly causing an unknown (undefined) hazard.

If design analyses (such as hazard analysis), requirements analysis, and software testing fail to expose unknown hazards, other ways to augment the set of known hazards are required. Fortunately, fault injection can play a role here.

# 2 Notation

When program $P$ executes on test case $I$, a series of state transitions, $S_{P1I}$, $S_{P2I}$, ..., are created until a program output $O_{PI}$ is produced for $I$. Thus the sequence of computations in $P$ creates a trace of states caused by internal and external events:

$$I \longrightarrow S_{P1I} \longrightarrow S_{P2I} \longrightarrow S_{P3I} \ldots S_{PNI} \longrightarrow O_{PI}.$$

Note that $I$ will be a vector that includes information about the state of the system and not imply the input to $P$. $N$ represents the last internal program state created for $I$ before $P$ produces an output.

When fault injection is applied to an internal state during an execution trace, the sequence of event states is modified. For example, suppose that we opt to corrupt the second state that is created during execution using $I$. Then we will have this sequence:

$$I \longrightarrow S_{P1I} \longrightarrow \boxed{S_{P2I} \longrightarrow S_{P2I'}} \longrightarrow S_{P3I'} \ldots S_{PN'I'} \longrightarrow O_{PI'},$$

where $S_{P2'}$ is the data anomaly created by fault injection. Since each successive state created after $S_{P2I'}$ may be different than the state that would be there had the data anomaly not been injected, we denote each of these states with the $'$ symbol. Note that $N'$ may or may not equal $N$.

What we have done here is to forcefully modify $S_{P2I}$ into $S_{P2I'}$. Thus we may have also forced each successor state (including the output state) to also be modified. Further, it is often the case that this state modification to $S_{P2I}$ will result in different output states, i.e., $O_{PI} \neq O_{PI'}$. When this occurs, the question that must be asked is: What can be gleaned from artificially corrupting $S_{P2I}$? That is, what have we learned about $P$'s behavior as a result of injecting $S_{P2I'}$? After all, that is all that we care about.

# 3 Three Categories

One of three situations is true as a result of artificially creating $S_{P2I'}$:

1. There exists an input $M$ from $P$'s domain that can naturally produce $S_{P2I'}$ or some successor $S_{PkI'}$, where $2 < k \leq N'$. Therefore, $O_{PM}$ is an output that the program can naturally produce.

(a) If $O_{PM}$ is already known to be an undesirable output from the hazard analysis, then $P$ needs to be made more robust to disallow $O_{PM}$. This means that $P$ needs better recovery capabilities.

(b) If $O_{PM}$ is not already known to be an undesirable output, then take another look at $O_{PM}$ to double check. Unimagined hazards can be found this way. This is *hazard mining.*

2. There does not exist an input $M$ from $P$'s domain that can naturally produce $S_{P2I'}$ or some $S_{PkI'}$, where $2 < k \leq N'$. But $S_{P2I'}$ causes $P$ to output an $O_{PI'}$ that can be produced by a different input $X$ that *is* part of the domain of the software.

   (a) If $O_{PX}$ is already known to be an undesirable output from the hazard analysis, then $P$ needs to be made more robust to disallow $O_{PX}$.

   (b) If $O_{PX}$ is not already known to be an undesirable output, then take another look to double check.

3. There does not exist an input in $P$'s domain that can naturally create $S_{P2I'}$, $S_{PkI'}$ (where $2 < k \leq N'$). And there does not exist an input in $P$'s domain that can naturally create $O_{PI'}$. Therefore, $O_{PI'}$ is a nonsense output event and no additional design-for-safety steps are warranted.

The phenomenon where two unique states go into an operator and result in the same state is the reason that the first two scenarios are separated. It is possible that there exists an input to $P$ that cannot create any state in the sequence $S_{P2I'} \ldots S_{PN'I'}$ but can result in $O_{PI'}$.

## 3.1 Discussion

Category 1(a) is the ideal situation but only if we get this information before the system is deployed. Here, an input $M$ from $P$'s domain exists that can naturally produce an undesirable output. If $M$ is not selected during testing of $P$, then we may never know that this is possible. Here, we will deploy a software system with the $O_{PM}$ software hazard lurking.

But at least fault injection forewarns us that $O_{PI'}$ is possible without knowing $M$ (here, $O_{PI'} = O_{PM}$). If we fear that $M$ might exist after having observed $O_{PI'}$, then additional *fault tolerance* mechanisms are warranted to ensure that $O_{PM}$ does not occur.

Category 1(b) is the situation where $O_{PI'}$ does not appear to be an event defined in the hazard analysis as hazardous. Recall that hazard analysis is performed at the system level, not simply at the software level. Therefore, $I$ is not simply the input to $P$, but it also includes the state of the system at the time when $P$ was executed on $I$.

In situation 1(b), an opportunity is provided to potentially *augment* the list of defined hazards with $O_{PI'}$ if it is determined that $O_{PI'}$ is truly an event that should not be allowed

to occur when the system is in the state contained in $I$. Here, we are "mining" for hazards that have not yet been defined.

The reason that category 1(b) is so important is because hazard analysis often fails to completely define all hazardous events. Hazard analysis is simply the process of trying to define up front all of the types of bad output events that must be protected against before a system is built. The process of defining hazards involves much human judgment, and clearly humans can inadvertently fail to define software hazards by not recognizing that they are possible.

Thus category 1(b) involves outputs that were not imagined. If a human who understands the system is willing to review those output events that do not satisfy the current set of defined hazards, then it is possible that the known set can be increased to a more complete listing of software hazards. It is true that human judgment can just as easily fail during this task as it can during the initial hazard analysis. However the key advantage here is that a human is simply asked to make a Boolean decision concerning whether an event is a hazard as opposed to attempting to build a list of all hazards from scratch.

As for categories 2 and 3, the same recommendations given above for categories 1(a) and 1(b) should be undertaken for categories 2(a) and 2(b). The third situation may cause unnecessary concern over the degree of safety afforded by the software. In the third situation, fault injection forced a software output to occur that is impossible under any natural circumstance. Here, we do not learn anything meaningful about how badly the software may behave in the future given $P$'s current domain. The key issue then is how often will fault injection put us into the third situation? If the answer is often, then fault injection is problematic. The goal is to fine-tune the state corruption process such that category 3 situations are unlikely. How best to do this is an open research issue.

The issue of how many artificial data anomalies to inject is also an open research issue. Quite honestly, there is no correct answer that fits all systems. The answer is dependent on resources available for a particular system, mainly, how many outputs is it reasonable to expect that can a person sift through in search of additional hazards? The answer to this will be a function of the complexity of the output of the system. Once the degree of available resources is determined, then the answer for how much fault injection to perform is easier to determine.

# 4 Results from One Experiment

As mentioned earlier, pseudo-random number generation is an integral process employed by fault injection. We will now look at the results from applying the SafetyNet tool (Reliable Software Technologies, Sterling, VA) to a transportation system. While a user of SafetyNet can customize their data anomalies, most users prefer to allow the tool to simply create them using the tool's built-in random number generator utility [7]. In the case of this transportation system, no custom data anomalies were used.

Our interest is in seeing how many output events from the software fell into which categories. Fault injection was applied to new control software proposed for the Bay Area Rapid Transit System. Twenty-six potential problems were reported after fault injection was performed. Twelve were Category 3 while fourteen were category 2. The developer of the software did not provide us with a further breakdown between categories 2(a) and 2(b), however we do know that several new hazards were discovered.

The reason that this is interesting is that randomly generated data anomalies were the only type injected into the control software. Yet slightly over half of the anomalies revealed that the software could produce undesirable outputs. This provided a road map into the software with an accompanying plan for how to best add fault tolerance mechanisms [3].

# 5 Conclusions

We have discussed two means for improving safety: (1) making the software more robust via additional fault tolerance mechanisms after the software is observed to cause hazards during fault injection, (2) by augmenting the list of defined hazards. The first technique is a specialized way to test the current level of safety afforded by a software program. The second technique is a way to test the thoroughness of the original system hazard analysis. These two techniques are unique among the current state-of-the-practice in the software safety field.

We do not mean to imply that fault injection will always be as successful at uncovering lurking problems as it was for the transportation system. There will be cases undoubtedly where no new hazards are discovered and seemingly little value was derived. This will certainly be true when the original hazard analysis was thorough. But then that is not a slight on fault injection, and is instead an acknowledgment of the success of the early requirements and specification effort. For example, when you go to your doctor and get a clean bill of health, that does not necessarily mean the doctor was negligent. Maybe you really are healthy.

# References

[1] B. H. Harvey, Editor. European Major Hazards, Oyez Scientific and Technical Services, Ltd., London, 1984. .

[2] R. L. Bates. *Industrial Minerals: How They Are Found and Used.* Enslow Publications, Inc., 1988.

[3] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, 14(4):73–83, July 1997.

[4] M. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[5] J. St. John. *Noble Metals.* Time-Life Books, 1984.

[6] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1998.

[7] S. K. Park and K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.