Comprehensive and Efficient Runtime Checking in System Software through Watchdogs

Chang Lou
Johns Hopkins University

Peng Huang
Johns Hopkins University

Scott Smith
Johns Hopkins University

ACM Reference Format:
Chang Lou, Peng Huang, and Scott Smith. 2019. Comprehensive and Efficient Runtime Checking in System Software through Watchdogs. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321440

Abstract

Systems software today is composed of numerous modules and exhibits complex failure modes. Existing failure detectors focus on catching simple, complete failures and treat programs uniformly at the process level. In this paper, we argue that modern software needs *intrinsic* failure detectors that are tailored to individual systems and can detect anomalies within a process at finer granularity. We particularly advocate a notion of intrinsic software *watchdogs* and propose an abstraction for it. Among the different styles of watchdogs, we believe watchdogs that *imitate* the main program can provide the best combination of completeness, accuracy and localization for detecting gray failures. But, manually constructing such mimic-type watchdogs is challenging and time-consuming. To close this gap, we present an early exploration for automatically generating mimic-type watchdogs.

1 Introduction

Software inevitably fails. In programs designed for interactive usage, a user can observe anomalies in the software and react. For example, the user of a text editor may find that the backspace key can no long delete characters so she may restart the process and restore the last saved file. But a large fraction of software today is a component of some online service that demands high availability with minimal manual intervention [13, 29]. Such software needs to proactively check whether it is functioning properly at runtime and autonomously take corrective action when serious problems are detected, and this requires effective failure detectors.

Despite their importance, failure detectors have traditionally been implemented as a thin module with simple logic *independent* of the underlying code: a monitored process is assumed to be working as long as it does something periodically based on the contract with the external detector, *e.g.*, replies to pings, sends heartbeat messages, or maintains sessions. This works fine for fail-stop failures, but it cannot detect complex gray failures [23] including partial disk failures [31], limplock [17], fail-slow hardware [11, 20], and state corruption [16], which are common in large cloud infrastructure.

These failure detectors are overly generic, treating all monitored software as coarse-grained "nodes". The resulting failure modes are inherently too simple to represent the complex behavior of modern software. The recently proposed Panorama [22] attempts to address this limitation by converting any requester of a monitored process into a logical observer and captures error evidence in the request paths. While this approach can enhance failure detection, the observers cannot identify why the failure occurs or isolate which part of the failing process is problematic, making subsequent failure diagnosis painful and time-consuming [4]. Hierarchical spies as proposed in Falcon [27] has similar limitations.

In this position paper, we argue that system software needs *intrinsic* failure detectors that detect anomalies *within* a process at finer granularity. An intrinsic detector monitors internally for subtle issues specific to a process, *e.g.*, checking if a Cassandra background task of SSTable compaction is stuck. Importantly, an intrinsic detector could pinpoint the problematic code region along with the payload for diagnosing and reproducing production failures. This precise fault information could further help expedite recovery.

In particular, we advocate a promising yet under-explored type of intrinsic failure detector – intrinsic *watchdogs* – for system software. Watchdogs are widely used in embedded devices such as deep space probes whose operation environment can be remote and hostile due to extreme temperature, high-intensity radiation, *etc*. They are hardware circuits controlled by embedded software to automatically detect anomalies within the system, and reset the processor or microcontroller in response to failure. In the broader software domain, however, only a few mature systems have adopted the watchdog concept [1]. Even for these systems, the watchdog modules are designed in a shallow way (*e.g.*, to only periodically

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotOS '19, May 13–15, 2019, Bertinoro, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6727-1/19/05. . . $15.00
https://doi.org/10.1145/3317550.3321440

Table 1. Comparison of three abstractions: crash failure detector, error handler and watchdog.

Abstraction	Scope	Execution	Goal	Checks	Target	Analogy
Crash FD	Extrinsic	Concurrent	Inform another process about a process's status to benefit a group	Liveness	Protocol failure	Heartbeat
Error handler	Intrinsic	In-place	Mitigate a known error in a specific program point to prevent failure	Safety	Low-level error	Immune sys.
Watchdog	Intrinsic	Concurrent	Monitor overall software health and assess if software is functioning	Safety+Live.	Partial failure	Blood test

check some status code), which are in fact equivalent to extrinsic detectors. It is unclear what a truly intrinsic software watchdog should be like and how to systematically write it.

To shed some light on this under-explored type of intrinsic failure detector, we propose a unique software watchdog abstraction (§3.1), elucidate characteristics of a good watchdog (§3.2), and describe design principles for coding watchdogs (§3.3). Based on these design principles we believe that a particularly effective type of watchdog is one that *mimics* the main program. But manually writing such a watchdog is time-consuming and challenging to get right as it needs to simulate the key interactions of the main program without negatively impacting execution. To address this gap we present an early exploration of how customized, mimic-type watchdogs can be generated for a given piece of software (§4), and conclude with a discussion of challenges and future directions (§5).

2 Background and Motivation

A Watchdog Timer (WDT) [12] is an essential component found in embedded systems to detect and handle hardware faults or software errors. WDT's use internal counters that start from an initial value and count down to zero. When the counter reaches zero, the watchdog resets the processor. In a multi-stage watchdog, it will initiate a series of actions upon timeout, such as generating an interrupt, activating fail-safe states, logging debug information and resetting the processor. To prevent a reset, the software must keep "kicking" the watchdog, typically by writing to some control port.

To effectively use a watchdog, the program must do more than just kick the watchdog at regular intervals – the software should perform sanity checks [19], on *e.g.* the stack depth, program counter, resource usage, and status of the mechanical components before kicking the watchdog. It is also good practice to insert a flag at each important point of the main loop and check all flags at the end [30]. In a multitasking system, the watchdog should also try to detect errors such as infinite loops, deadlock, and priority inversion in each task.

At a high level, the basic hardware watchdog timer mechanism is similar to a crash failure detector [15] in distributed systems. But there are two main distinctions to be made. First, the watchdog's control logic is closely coupled with the internal software state, it is an *intrinsic* component; crash failure detectors on the other hand share little software state, they are *extrinsic*. Second, the primary goal for a failure detector in a distributed system is to benefit a group of processes. When a failing process is detected, the process group can use that information to adjust its behavior for *e.g.*, consensus, broadcasting, load balancing, or deciding a replica set. In comparison, the primary goal for a watchdog is to benefit the individual process it monitors, and a fault that does not immediately concern the process group protocol may still be of interest to a watchdog. Watchdogs and traditional failure detectors can and should co-exist. Watchdogs have better access to internal system states and thus will be particularly suitable for dealing with partial failures. Crash failure detectors have stronger isolation. So if severe thrashing causes an entire process to become unresponsive and jeopardizes the watchdogs as well, a crash detector can still catch the failure.

In contrast with the popularity of watchdogs in embedded systems, mainstream software engineering does not put much emphasis on designing intrinsic watchdogs despite a similar need for robustness. The primary way for system software to catch subtle failures today is to rely on error handlers or in-place assertions. Although error handlers do a good job of detecting some program-specific faults, they are designed as part of the main business logic, with the goal of mitigating some specific known error condition associated with *a specific program point* for the program to continue execution. So for example catching an `EINTR` error during a `write` system call is the target of an error handler, and it may retry successfully. Watchdogs on the other hand are tasked to monitor overall software health and give a definitive assessment as to whether the software is still functioning properly. In addition, complex failures such as metadata inconsistency and data integrity require complex checks beyond a single operation and are thus not suitable for error handlers. Watchdogs can run complex `fsck`-like checks in parallel to the normal execution to catch such failures. Liveness-related failures often also do not have explicit error signals that can trigger a handler: there is no signal for *e.g.*, `write` being blocked indefinitely or some thread deadlocking or infinitely looping. So, these sorts of failures are the target of watchdogs but not error handlers. Table 1 summarizes the comparison of crash failure detectors, intrinsic watchdogs, and error handlers.

3 Designing Watchdogs for Software

A primitive form of software watchdog can be realized through *ad-hoc* runtime checking. For example, seasoned developers may create a thread that periodically checks if enough memory remains. In this Section, we propose a watchdog design to systematically perform runtime checking, and describe principles behind the effective construction of software watchdogs.

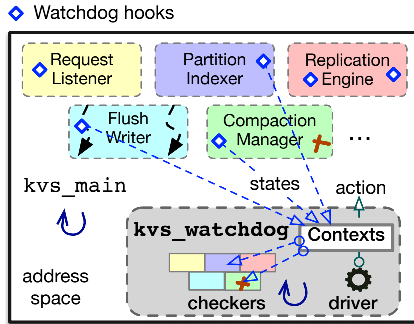

Figure 1. kvs running with its watchdog in production.

To make the discussion more concrete, we use a key-value store kvs as a running example. Despite its simple interface (GET, SET, APPEND, DEL), kvs has complex internals, including request listener, indexer, disk flusher, replication engine, *etc.*

3.1 A Watchdog Abstraction

Basic Structure. A watchdog is an extension embedded in the main program to monitor system status. It lives *within* the program's address space (Figure 1) and encapsulates all the checking procedures (*checkers*). Each checker stores a sequence of specific instructions tailored to inspect a certain part of the main program, such as the kvs indexer, for expected behavior. A watchdog driver will manage checker scheduling and execution. When a checker executes, it might get stuck, crash, or trigger an error. The watchdog driver catches failure signatures from checkers, aborts or restarts their executions and applies an action to the main program accordingly.

Concurrent Execution. It is natural to insert the watchdog checkers directly in the main program. However, adding many checkers in place can make the original code lengthy, hard to maintain and even introduce unintended effects. For example, checkers with an improper try-catch can alter the main execution flow unexpectedly, making the software fragile. Moreover, faults like control flow errors [9, 16, 28] and deadlocks fundamentally cannot be checked in place by the executing instruction. From a performance perspective, executing heavyweight checkers in-place also incurs a significant overhead on overall execution time.

We instead propose a design where the watchdog runs *concurrently* along with the main program. Concurrent execution allows checking to be decoupled so a watchdog can execute as many checkers as necessary to catch faults comprehensively, without slowing down the main program during fault-free execution. When a checker crashes or triggers an assertion because a fault was exposed, the concurrent watchdog also will not unexpectedly alter the execution of the main program.

State Synchronization. If we are not careful, concurrent checkers might report failures that do not exist in the main program. For example, if kvs was configured to be in-memory only or if the snapshot directory does not exist yet because no SET requests are received, an error report from the disk flusher checker would be spurious. To address this issue, the watchdog must synchronize state with the main program. We introduce the notion of *context* for this purpose. A context is bound with each checker and supplies the payload and arguments to the checking procedures. State synchronization is achieved through watchdog hooks placed in the main program. When the main program execution reaches the hook points, the hook uses the current program state to update the context in the watchdog. The watchdog driver will ensure that a checker's context is ready before executing it. To avoid altering main execution, this synchronization is *one-way*.

3.2 What Makes for a Good Watchdog?

With the watchdog extension running together with the main program, a process will have its normal execution and the checking execution. In a poorly designed watchdog, the checking execution is too *disjoint* from normal execution: it will create a false sense of health even when the main program is broken for a long time or it will "bark" excessively even when the main program executes smoothly. In contrast, a good watchdog *intersects* the normal execution so it accurately reflects the status of the main program. Statically, this means the checkers of a good watchdog achieve high coverage of both the main program's major components and the variety of safety and liveness requirements these components may potentially violate. Dynamically, a good watchdog's contexts are in sync with the main program states.

With these two executions, a good watchdog must provide strong isolation. Executing watchdog checkers should not incur unintended side-effects or add significant cost to the normal execution. For example, in monitoring the indexer of kvs, the checkers may try to retrieve or insert some keys, which should not overwrite data produced from the normal execution or significantly delay normal request handling. The isolation also applies in the opposite direction: a bug in the main program should not compromise the whole watchdog.

3.3 How to Write Watchdog Checkers?

Principles. Because a good watchdog should accurately reflect the main program's status without imposing on the normal execution, we believe the first design principle in writing watchdog checkers is to *let the checker share a similar fate as the main program*. For example, if the kvs replication engine is stuck in waiting for an incoming message, a corresponding checker may also hang in executing some polling test (but, the watchdog driver itself is not affected and can detect the checker hang). As another example, to detect memory pressure in a Java program, a checker can run a worker thread in a loop sleeping for a short time; if when the worker awakens, the elapsed time is significantly larger than the specified sleep time, the checker likely suffered from a long GC pause [2].

Table 2. Three types of watchdog checkers. ✘: it cannot pinpoint the cause of failure; ✦: it can narrow down causes to some extent; ✔: it can pinpoint the error.

Type	Level	Example	Completeness	Accuracy	Pinpoint
Probe	API	App spy, httpd mod_watchdog	Weak	Perfect	✘
Signal	Resource	WDT, Linux watchdogd	Modest	Weak	✦
Mimic	Operation	HDFS disk checker (partly)	Strong	Strong	✔

This implies that the main program is likely experiencing excessive memory usage or a serious memory leak [6].

In addition, checkers should *operate at a level close to the failure*. Checkers that only monitor client-facing interface functions can easily miss detection of many internal faults (*e.g.*, silent failure in a compaction background task). They also make diagnosis and reaction to a reported failure difficult. On the other hand, it is hard to require checkers to always pinpoint the exact location of the bug. A more realistic goal is to have the checkers report failure at a location that is in the ballpark of the root cause, *e.g.*, several instructions away in the same function, or at caller of the faulting function.

While it is tempting to check as much as possible, *checkers should focus on catching faults manifestable only in a production environment*. For example, since kvs partitions may be corrupted in production due to either hardware problems or unexpected code bugs, it is worthwhile to have a checker that computes and validates the checksum of each partition. Logically deterministic errors that lead to wrong results, on the other hand, are better eliminated through unit testing. For example, the kvs partition manager is supposed to sort the key ranges in all partitions in ascending order. Relying on a watchdog checker to ensure this correctness property may be an excuse for laziness in quality assurance. Of course, testing cannot eliminate all logic bugs, so it is possible that the partition manager in production can sort keys incorrectly in some corner cases. If resource is not a concern, we can afford to run a correctness checker to catch these corner cases. But in reality, we need to prioritize checking with limited resources. Such correctness checking would yield diminishing returns.

Approaches. There are three general approaches to checker construction (Table 2). The simplest one is a probe-based checker which acts like a special client and invokes the software's public APIs with pre-supplied input. For example, a checker for kvs can keep submitting SET and GET requests and check if the requests succeed; this approach resembles application spies [27], observers [22] or Apache mod_watchdog [1]. The accuracy of a probe checker is perfect: any error it detects is a true violation of the contract the software provides. However, the probe checker suffers from incompleteness: if the program has numerous public APIs, it will be challenging to cover all of them. Even for a program with few public APIs like kvs, since probe checkers use some pre-supplied input, they can miss many corner cases. For example, kvs may fail to handle requests with a key size larger than 64 B or multiple SET's followed by a GET; or, kvs may fail to write/replicate/-compress the data even when it returns success from a SET request. Another weakness of the probe checker is its inability to localize what might cause the failure (*e.g.*, timeout of SET).

The second approach is to define some system health indicators and then write a checker to monitor each one. The Linux watchdog daemon [5] is such an example. Its checkers monitor whether the process table is full, important files are accessible, certain IP addresses can be pinged, the load average is too high, *etc.* The signal checker is good at detecting environment/resource faults. But its accuracy is weak. For example, when the checker finds kvs's request queue is full or the memory usage is high, kvs might in fact be processing a continuous stream of requests without error. Writing such checkers thus requires knowing failure patterns beforehand and needs significant tuning to be accurate.

In the third approach, a checker selects important operations from the main program, mimics them and detects errors. Since the mimic checker exercises similar code logic in a production environment, it can catch both faults external to the program (*e.g.*, bad network, low free memory) and defects in the software. For example, the disk checker module in HDFS initially only checked directory permissions, but later it was enhanced [3] to create some files and invoke functions from the DataNode main program to do real I/O in a similar way. Besides detection, the mimic checker can pinpoint the failing instruction and error information. The challenge with this approach is how to systematically select operations from the main program and mimic them in the checker.

A system can design all three types of watchdogs in combination to complement each other. Probe and signal watchdogs are lightweight and easy to construct. Mimic watchdogs are powerful but require deeper domain knowledge about the system to write. Mimic watchdogs can also benefit from the end-to-end view of probe watchdogs. So when a mimic watchdog detects a potential partial failure, it can invoke a probe checker to validate the impact of the failure.

4 Toward Generating Watchdogs

It is time-consuming for developers to manually write good watchdogs, and it is challenging to get it right – *e.g.*, incautiously written watchdogs can miss checking important modules, use inconsistent checking policies, alter the main execution, invoke a dangerous operation, corrupt main data structures, *etc.* Moreover, the watchdog needs to be kept consistent with the main program as the software evolves. These problems are particularly acute in the desirable mimic-type

```
1  public class SyncRequestProcessor {
2   public static void serializeSnapshot(DataTree dt, ...) {
3     ...
4     dt.serialize(oa, "tree");   keep reducing
5   }
6 }
7 public class DataTree {
8   public void serialize(OutputArchive oa, String tag) {
9     scount = 0;
10    serializeNode(oa, new StringBuilder(""));  keep reducing
11    ...
12  }
13  public void serializeNode(OutputArchive oa, ...) {
14    DataNode node = getNode(pathString);
15    if (node == null)
16      return;
17    String children[] = null;
18    synchronized (node) {               locate vulnerable operation
19      scount++;
20      oa.writeRecord(node, "node");
21      children = node.getChildren();
22    }
23    path.append('/');
24    for (String child : children) {
25      path.append(child);
26      serializeNode(oa, path); // serialize children
27    }
28  }  + ContextFactory.serializeSnapshot_
29 }      reduced_args_setter(oa, node);
       insert watchdog hooks
```

Figure 2. Reducing a code snippet from ZooKeeper.

```
1  public class SyncRequestProcessor$Checker {
2    public static void serializeSnapshot_reduced(
3         OutputArchive arg0, DataNode arg1) {
4      arg0.writeRecord(arg1, "node");
5    }
6    public static void serializeSnapshot_invoke() {
7      Context ctx = ContextFactory.
8         serializeSnapshot_reduced_context();
9      if (ctx.status == READY) {
10       OutputArchive arg0 = ctx.args_getter(0);
11       DataNode arg1 = ctx.args_getter(1);
12       serializeSnapshot_reduced(arg0, arg1);
13     }
14     else
15       LOG.debug("checker context not ready");
16   }
17   public static Status checkTargetFunction0() {
18     ...
19     serializeSnapshot_invoke();
20     ...
21   }
22 }
```

Figure 3. A checker generated for Figure 2.

watchdogs as they are mimicking the original program. To address these issues, we propose a method using static analysis to automatically generate mimic-type watchdogs that follow the principles described in Section 3. The core technique behind this method we call *program logic reduction*.

4.1 Program Logic Reduction

Given a program P, we want to create a watchdog W that can detect gray failures in P without imposing on P's execution. One approach is to extract a set of program slices [33] of P that are involved in handling each type of API request, and periodically execute them in W. But such slices would include substantial portions of P in practice, which are not only heavyweight but also challenging to pinpoint faults.

We instead propose to derive from P a reduced but representative version W, which nonetheless retains enough code to expose gray failures. Our hypothesis that such reduction is viable stems from two insights. First, most code in P need not be checked at runtime because its correctness is logically deterministic – such code is better suited for unit testing before production and thus should be excluded from W. Second, W's goal is to catch errors rather than to recreate all the details of P's business logic. Therefore, W does *not* need to mimic the full execution of P. For example, if P invoked `write()` many times in a loop, for checking purposes, W may only need to invoke `write()` once to detect a fault.

We design the reduction algorithm as follows. First, we extract code regions that may be executed continuously. In this way, we exclude checking for code execution in the initialization stage. Multiple long running regions may be identified. For example, in ZooKeeper, it could include regions in the replication workflow, the request processor, and the snapshot service. Then for each such code region, we are interested in only retaining operations that are worthy of monitoring. Our criteria for selecting such operations are those that are vulnerable to fail in production due to either environment issues or bugs, such as I/O, synchronization, resource, and communication related method invocations. We also support annotations for developers to tag customized vulnerable methods. We then construct a checker C by extracting all vulnerable operations in a function, removing similar vulnerable operations, and performing a global reduction along the call chains.

C at this point cannot be directly executed, however, due to uninitialized variables or parameters. So we further analyze the *context* required for the execution of C. A context factory with APIs for W to manage the dependent context of C will be generated. We then enhance C with runtime checks based on the extracted operations to detect both safety and liveness violations. Finally, we insert context API hooks in P to synchronize state.

4.2 Preliminary Results

We have built a prototype, *AutoWatchdog*, that implements the above watchdog generation method. Its core component is written on top of the Soot [32] framework and thus only supports Java bytecode-based software. But the proposed technique is not Java-specific. AutoWatchdog provides a generic watchdog driver and checker recipes for scaffolding. Developers can optionally specify in the configuration what types of system-specific operations might be vulnerable. AutoWatchdog will try to locate these operations when performing the program reduction. All the generated checkers will be added to the watchdog driver, which manages the checker executions at runtime. In the end, AutoWatchdog instruments the main program with the watchdog hooks and packages the watchdog driver including the checkers into the original software.

We have been able to successfully apply AutoWatchdog to three pieces of large-scale real-world system software –

ZooKeeper, Cassandra and HDFS – and generate tens of checkers for each. Figure 2 shows an example from ZooKeeper that AutoWatchdog analyzed. In reducing the `serializeSnapshot` function, AutoWatchdog keeps following the callees and determines line 20 is potentially vulnerable. As shown in Figure 3, AutoWatchdog eventually generates a reduced version of the `serializeSnapshot` function that retains the identified vulnerable operation, along with a checker that invokes the reduced function. In addition, AutoWatchdog instruments a hook between line 19 and line 20 in Figure 2 to prepare context for the reduced function.

We reproduced a real-world gray failure in ZooKeeper [7] to test the generated checkers. In this example, a network issue causes a remote sync to block in a critical section, hanging all write request processing. ZooKeeper's heartbeat detection protocol and admin monitoring command [8] both showed the faulty leader as healthy during the entire failure period, whereas our generated watchdog detected the timeout fault in around seven seconds and pinpointed the blocked function call with a concrete context.

5 Discussion

5.1 Challenges

We now describe several open challenges in automating mimic-type watchdog construction. Our analysis to detect what operations are vulnerable requires some domain knowledge, and to fully automate this analysis remains a challenge. Currently, we catch failure signatures from a reduced code snippet through generic checks based on the types of operations. This works well for liveness issues and common safety violations, but the watchdog could benefit from incorporating more semantic checks. The watchdog detection may also be superfluous if the main program can successfully handle the detected fault. To reduce false alarms, we need to further assess the impact of the fault, *e.g.*, through invoking probe-checkers when mimic-checkers detect faults. The isolation of watchdogs also needs to be further enhanced. We currently can prevent memory side effects with a context replication mechanism and common I/O side effects with a redirection mechanism; however, they are imperfect for complex I/O patterns. Sharing the same address space with the main program also means a wild pointer in main program may subvert the watchdog. To prevent this would require a memory protection mechanism for the watchdog.

5.2 Opportunities

Cheap Recovery. Rebooting is often an effective recovery option, especially for non-deterministic failures, but full restart is expensive. With the detailed localization information from our watchdogs, we can expedite recoveries by replacing corrupted objects, threads, files, *etc.* to quickly restore the service. Techniques like microreboot [14] can potentially integrate well with watchdogs.

Failure Reproduction. It is notoriously difficult to reproduce production failures [35]. Since mimic-type watchdogs not only isolate the faulty code regions but also capture the failure-inducing context (*e.g.*, a corrupt message), developers can leverage the recorded information for failure reproduction and postmortem analysis.

6 Related Work

Failure detectors have been extensively studied (*e.g.*, [10, 15, 25–27]), but they primarily focus on externally detecting fail-stop failures in large distributed systems. We focus on detecting and localizing partial failures within a process. Several projects attempt to generate checks for software. Daikon [18] infers likely invariants of a program from its dynamic execution traces. InvGen [21] uses a combination of static and dynamic analysis to generate invariants. These checks are mainly for in-place assertions. AutoWatchdog targets generating concurrent checkers for intrinsic watchdogs, and it could benefit from these semantic checks. PCHECK [34] extracts configuration usage with program slicing to detect latent misconfiguration during initialization. We focus on synthesizing checkers for monitoring long-running procedures of a software in production by using a novel program reduction technique. Failure sketching [24] uses slicing and control flow tracking to generate a sketch for a given failure to ease debugging, which is complimentary to our proposed technique and goal. Hardware designers have explored using a watchdog co-processor to concurrently detect control-flow and memory access errors in the main processor [28]. Our concurrent error detection scheme is inspired by this classic design. We explore this scheme in the software domain where the watchdog is intrinsic to the main program and needs to catch a variety of safety and liveness violations.

7 Conclusion

As systems software becomes increasingly complex and fails in ever more subtle ways, there is a strong need for intrinsic, customized failure detectors – watchdogs. We propose an abstraction for robust concurrent watchdog design. Among several flavors of watchdogs, mimic-type watchdogs are particularly powerful, but writing them is challenging. To address this gap, we propose a method that analyzes the source code of a given program and automatically generates mimic watchdogs by reducing the main program. Our preliminary results show the approach to be promising for real-world software.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful reviews. We also thank Chuanxiong Guo and Lidong Zhou for early discussion on the motivation of watchdogs for system software.

References

[1] Apache module mod_watchdog. https://httpd.apache.org/docs/2.4/mod/mod_watchdog.html.

[2] Detecting long jvm GC pause detector in Ignite. https://issues.apache.org/jira/browse/IGNITE-6171.

[3] HDFS disck checker. https://issues.apache.org/jira/browse/HADOOP-13738.

[4] Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

[5] Linux watchdog daemon. https://linux.die.net/man/8/watchdog.

[6] Memory leak in HBase. https://issues.apache.org/jira/browse/HBASE-21228.

[7] Network issues can cause cluster to hang due to near-deadlock. https://issues.apache.org/jira/browse/ZOOKEEPER-2201.

[8] Zookeeper administrator's guide. https://zookeeper.apache.org/doc/r3.4.8/zookeeperAdmin.html.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, Alexandria, VA, USA, 2005. ACM.

[10] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, Monte Verità, Switzerland, May 2009. USENIX Association.

[11] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01. IEEE Computer Society, 2001.

[12] A. S. Berger. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books. Taylor & Francis, 2001.

[13] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[14] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, San Francisco, CA, 2004. USENIX Association.

[15] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[16] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Boston, MA, 2012. USENIX Association.

[17] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, 2013. ACM.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, Los Angeles, California, USA, 1999. ACM.

[19] J. Ganssle. Great watchdog timers for embedded systems. http://www.ganssle.com/watchdogs.htm.

[20] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, USA, 2018. USENIX Association.

[21] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 634–640, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018. USENIX Association.

[23] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, 2017. ACM.

[24] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, Monterey, California, 2015. ACM.

[25] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, 2013. USENIX Association.

[26] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, Bordeaux, France, 2015. ACM.

[27] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the Twenty-third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, 2011. ACM.

[28] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[29] J. C. Mogul, R. Isaacs, and B. Welch. Thinking about availability in large service infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 12–17, Whistler, BC, Canada, 2017. ACM.

[30] N. Murphy. Watchdog timers. *Embedded Systems Programming*, pages 112–124, 2000.

[31] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, Brighton, United Kingdom, 2005. ACM.

[32] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

[33] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[34] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 619–634, Savannah, GA, USA, 2016. USENIX Association.

[35] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 19–33, Shanghai, China, 2017. ACM.