Selective Symbolic Type-Guided Checkpointing and Restoration for Autonomous Vehicle Repair

Yu Huang
University of Michigan
yhhy@umich.edu

Kevin Angstadt
University of Michigan
angstadt@umich.edu

Kevin Leach
University of Michigan
kjleach@umich.edu

Westley Weimer
University of Michigan
weimerw@umich.edu

ABSTRACT

Fault tolerant design can help autonomous vehicle systems address defects, environmental changes and security attacks. Checkpoint and restoration fault tolerance techniques save a copy of an application's state before a problem occurs and restore that state afterwards. However, traditional Checkpoint/Restore techniques still admit high overhead, may carry along tainted data, and rarely operate in tandem with human-written or automated repairs that modify source code or alter data layout. Thus, it can be difficult to apply traditional Checkpoint/Restore techniques to solve the issues of non-environmental defects, security attacks or software bugs. To address such challenges, in this paper, we propose and evaluate a selective checkpoint and restore (SCR) technique that records only critical system state based on types and minimal symbolic annotations to deploy repaired programs. We found that using source-level symbolic information allows an application to be resumed even after its code is modified in our evaluation. We evaluate our approach using a commodity autonomous vehicle system and demonstrate that it admits manual and automated software repairs, does not carry tainted data, and has low overhead.

KEYWORDS

checkpoint, restore, repair, maintenance, autonomous vehicle

ACM Reference Format:
Yu Huang, Kevin Angstadt, Kevin Leach, and Westley Weimer. 2020. Selective Symbolic Type-Guided Checkpointing and Restoration for Autonomous Vehicle Repair. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3392201

1 INTRODUCTION

Autonomous vehicle systems (AVS) are an emerging area facing many software engineering challenges [57]. These systems must handle failures caused by software bugs, environmental changes, and security attacks. Addressing such issues usually requires costly human intervention and deployment of system repairs. Previous work in fault tolerant systems has successfully reduced human effort for more traditional server and workstation software settings [18, 38, 45]. One important aspect of fault tolerance is *failure transparency* [27], or the extent to which failures are invisible to users and applications. Failure transparency is especially relevant for cyber physical systems (such as AVS) that interact with the physical world: a vehicle may continue to move through inertia or gravity even if its software systems fail, pause or restart. To provide failure transparency, applications must resume after failures without help from users and also minimize space or time overhead compared to failure-free execution. One common approach, called *Checkpoint/Restore* or *Checkpoint/Restart*, provides such transparency by checkpointing the program state at a known-good configuration and then resuming later if there is a failure.

There has been significant interest in mitigating software failures in modern embedded systems, especially those with a high requirement for resiliency, such as AVS [17, 64]. Compared to traditional systems, AVS face more issues associated with rich environmental inputs and interactions. This complicates failure transparency compared to server or workstation software systems [16].However, most Checkpoint/Restore systems depend on the "fail-stop assumption" [51] (applications will not commit faulty state) and do not allow deployment of patches in the restoration phase, which is not applicable to certain autonomous vehicle software bugs or security attacks. In this paper, we propose to address three challenges related to Checkpoint/Restore to provide failure transparency that admits the deployment of software repairs for AVS.

First, Checkpoint/Restore must *support patches* to software. AVS require rapid repair of system defects and vulnerabilities. Traditional Checkpoint/Restore systems are based on the assumption that unexpected environmental conditions (e.g., dropped network packets and power outages) are transient and will not be present when the system is restored [51]. However, non-environmental defects are rarely transient. For example, a null pointer dereference resulting from faulty logic will usually recur if the software system resumes. To address such issues, the system must be patched or updated. This requires that a different (patched) system is restored from the one that was checkpointed. As a result, Checkpoint/Restore must be able to operate in cases where a software's code and related data structures are changed from the application of system repairs between the checkpointing and restoration.

Second, Checkpoint/Restore must be *low overhead*. For real time embedded systems like AVS, there is very limited storage space and tolerance for time overhead. Traditional Checkpoint/Restore solutions introduce significant space, time and hardware costs. Early techniques often required redundant hardware to mirror ongoing computations [39]. Even modern user-space techniques that do not require hardware support still suffer from high space overhead [1]. Moreover, Checkpoint/Restore must be able to complete before timing constraints are violated. In the case of AVS, violating a timing

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392201

constraint with high-overhead checkpointing anf restoration may cause the vehicle to become physically damaged.

Third, Checkpoint/Restore must not carry along *tainted data*. AVS must be able to resume without faulty behaviors recurring. Traditional Checkpoint/Restore solutions often struggle with security attacks that taint application state. This concern relates to the abstraction of failure transparency [27], which requires two invariants: sufficient application state must be saved to guarantee the application can resume from before the failure occurs, and sufficient application state must be lost so that the application can recover from failures affecting the state. Saving either too much or too little application state precludes failure transparency. Since most traditional Checkpoint/Restore approaches store the entire system state, they also store attacker-tainted data, resulting in post-restoration systems that remain compromised.

In this paper, we combine two insights to address these challenges and limitations. Our first algorithmic insight is to leverage existing developer domain knowledge for symbolic annotations to specify the critical state of an application. Modern object-oriented and imperative software systems tend to encapsulate related variables in classes or structures. This includes AVS software, such as ROS[1]or ArduPilot.[2] As a result, a small number of type-based annotations can often precisely capture the most critical state of a system (e.g., state that is too expensive to recompute or must otherwise be carefully checkpointed to allow the system to restore correctly). Our second insight is that we can automatically generate symbolic application-specific Checkpointing/Restore code in a type-directed manner. The use of symbolic references allows the system to resume in the face of most repairs (e.g., changes to program data structures need not preclude restoration). We thus present a selective checkpoint and restore (SCR) algorithm which saves the critical state of an application based on minimal annotations.

To evaluate SCR, we present the results of a case study focusing on its application to AVS software. To assess resuming a system in the presence of software repairs, we consider ArduPilot, a common AVS framework, as the specimen system. We use human-written patches to ArduPilot and also consider a patch produced by the Darjeeling automated program repair algorithm [58]. To evaluate efficiency, we measure the time and space overhead of resuming the AVS system and compare against CRIU [35], a state-of-the-art Checkpoint/Restore baseline. To evaluate whether the system can avoid checkpointing tainted data during security attacks, we evaluate SCR using a security attack provided by an independent third part (a “Red Team”). Finally, to assess usability, we measure the annotation effort required to deploy it in this AVS scenario.

In summary, the main contributions of this paper are:

- SCR, a Selective Checkpoint/Restore algorithm for recording and restoring critical parts of system state in a type-guided manner with minimal annotation burden and negligible space overhead.
- An evaluation of SCR's time and space overhead, resumption success, and annotation burden compared to CRIU, a state-of-the-art baseline, using human- and machine-generated patches for ArduPilot, an idicative AVS. SCR's time overhead is typically 2–20% higher than CRIU's, but its space overhead is 2000× lower.
- Empirical results measuring SCR's end-to-end efficiency and success using the ArduPilot AVS. SCR successfully checkpoints and resumes the system in the presence of both human-written and automated system repairs and tainted data in 100% of all considered scenarios, while CRIU only succeeds less than 40% of the time.

[1]http://www.ros.org — the Robot Operating System
[2]https://ardupilot.com — an open-source autopilot system

2 RELATED WORK

In this section, we discuss related work in Checkpoint/Restore systems, autonomous vehicle systems, software repair techniques, and manual annotations.

2.1 Checkpoint/Restore Systems

Checkpoint/Restore systems have been widely studied to provide fault tolerance and failure transparency. We discuss library and system implementations [49].

Library-based solutions often require an application to be linked with a Checkpoint/Restore library that provides necessary serialization primitives [1, 4, 6, 38]. Some library-based Checkpoint/Restore solutions, such as libckpt [61] allow user-directed checkpointing, which can improve the performance of checkpointing, but it does not apply to heap variables or to variables which reside in the statically-allocated data segment. Additionally, some programming environments provide serialization support, such as pickling combinators for functional programming [20] or serialization interfaces in the Boost library for C++ [44].

In contrast to library approaches, system-level implementations of Checkpoint/Restore mechanisms are usually built upon the operating system or hardware [11, 37, 49, 60, 65]. Recent system-level implementations introduce different improvements, such as support for parallel programming models and side effects [24, 32, 34, 36, 50]. CRIU [35] is a popular system-level approach mainly implemented in user-space for Linux. It can Checkpoint/Restore an application by saving the entire process tree. While CRIU is a mature, stable, widely-used tool, it does not admit resilience against changed or patched versions of software nor does it support selectively checkpointing a subset of a program's state. Dynamic Software Updating (DSU) [12, 52] can allow changing software at runtime (e.g., by altering function pointers), but still risks carrying over tainted data.

To the best of our knowledge, our proposed SCR approach is the first technique designed to support source code updates between checkpointing and restoration with low space and performance overhead. Supporting these goals requires that we make use of user-provided annotations, but our approach does not require extra library linking and will work on variables laid out in any data segment. Moreover, our SCR approach does not depend on a particular system environment because the annotations, checkpointing, and restoration implementations are confined to the application code without depending on specific kernel functionality. In Section 4.2, we articulate how SCR is applied to a real-world AVS scenario.

2.2 Security Assurance and AVS

Autonomous vehicle systems are complex systems that combine hardware for interfacing with sensors and actuators with control software designed to complete a statically- or dynamically-defined mission. A *mission*, or task, is a sequence of operations that the vehicle is directed to complete. In practice, missions often consist a sequence of GPS coordinates to which the vehicle must navigate.

Several researchers have discussed the security issues of autonomous vehicles [17, 22, 42, 56, 64], including embedded computing and sensors, policies, network connectivity, and attack and defense strategies. Well-studied security issues of autonomous vehicles include confidentiality, integrity and availability threats [16]. Malicious attacks can happen through communication interruption or the fabrication or modification of information, such as malware injection, GPS spoofing, and control channel jamming [29, 54]. Some widely-used system-level defenses include address space layout randomization [14], instruction set randomization [19], and the NX bit, among others. These approaches defend against different types of security attacks, such as buffer overruns and control flow hijacking. However, they cannot cover all potential attacks and are not helpful for large classes of defects. Because there are many security attacks that touch different data structures, we propose a symbolic approach focused on mission-critical data that avoids checkpointing other, possibly-tainted, information.

2.3 Automated Program Repair

Automated program repair techniques take a program and a test suite (or specification) as input and generate patches to make the program satisfy that notion of correctness. While we do propose novel automated program repair algorithms here, recent approaches have used automated program repair "on the fly" for autonomous vehicles [13, 62]. Automated program repair provides a strong motivation for enabling technology for resilient autonomous systems that checkpoint state, automatically synthesize and deploy software-level repairs mid-mission, restore state and resume the mission.

A significant amount of research has shown that automated program repair can fix bugs for real-world software [7, 25, 26, 28, 30, 33, 40, 41, 59, 63], scaling to millions of lines of code and thousands of test cases. A thorough discussion of automated program repair is beyond the scope of this work; [9] present a survey and [31] gives an annotated bibliography of recent advances. Traditional Checkpoint/Restore techniques do not support source code changes during restoration. Our symbolic approach to checkpointing, based on source-level variable names, supports combining program patching (manual or automatic) with Checkpoint/Restore to defend against attacks and software defects in a system.

2.4 Annotations

In our proposed SCR algorithm, we rely on symbolic annotations provided by the user to indicate the critical state to Checkpoint/Restore. In software engineering, various approaches have been proposed to reduce such annotation effort. Houdini [8] is an automatic tool to reduce the cost of writing specifications manually by generating a large number of candidate annotations and using axiomatic semantics to retain only those that correctly describe the system. The annotations inferred by Houdini describe functional behavior but do not pinpoint security-critical data.

A number of modeling or specification languages, including SLIC [2], Z [55], and Alloy [15], have been used to describe a system's abstract behavior. These are not a natural fit for our approach because they are designed to abstract away implementation details of a software whereas we need implementation information and relevant data structures to guide selective checkpointing.

CQual [53], JQual [10] and Banshee [23] are type qualifier systems that can be used to detect tainting errors (e.g., format string bugs). These techniques could work for annotating critical state, but with certain caveats. First, they tend to focus on specific security vulnerabilities (e.g., related to the sources and sinks of a given tainting model). More importantly, they typically attempt to compute a minimal set of possibly-tainted variables. Checkpointing everything not marked as possibly-tainted would incur too much overhead in our AVS use case.

3 AUTOMATIC SELECTIVE CHECKPOINT AND RESTORE GENERATION

In this section, we present SCR, an algorithm for selectively and symbolically checkpointing system state in a type-guided manner, as specified by manual annotations. To use SCR, a user first annotates particular variables (see Section 3.2) that should be (or should not be) checkpointed as well as program points at which that checkpointing should occur. Then, SCR automatically generates serialization and deserialization functions that checkpoint and restore those selected variables symbolically (i.e., based on their source-level names, not on their addresses, offsets or layouts) and links this code with the system to be protected (see Section 3.1). These functions carry out a recursive, type-guided traversal of relevant data structures (with appropriate memoization to handle cyclic references). At run-time, checkpoints are periodically created, and after a patch or other software modification, the last checkpoint (taken with respect to the pre-patch software) is restored.

3.1 SCR serialization and deserialization

The heart of SCR is thus the automatic generation of serialization and deserialization functions. This process is selective (i.e., it only considers variables implicated by user annotations) and symbolic (i.e., named components of complex data structures are stored individually, rather than as contiguous blocks of memory). This aspect of SCR is similar in spirit to pickling combinators or serialization libraries [20]. However, pickling combinators depend on functional language features (e.g., functions are explicitly first class), and serialization libraries require linking against potentially large libraries that do not admit adoption in embedded devices (e.g., AVS).

Algorithm 1 provides pseudocode for serialization code generation in SCR. Critically, it operates both at compile- (processing types and emitting serialization code) and run-time (executing the generated serialization code to create checkpoints). For every source-level type τ that is relevant to serialization (as determined by the annotations), a corresponding `serialize_`τ`()` function is created at compile time. If type τ_1 contains a (recursive) reference to type τ_2, then `serialize_`τ_1`()` contains a (recursive) call to `serialize_`τ_2`()`.

Given a program P, a variable x of type τ to serialize, and a source location l, the algorithm generates serialization functions at compile time by recursively decomposing the structure of the type

Algorithm 1 High-level pseudocode for SCR.

```
Input: Program $P$ to be selectively checkpointed.
Input: A variable $x \in P$ of type $\tau$ to be checkpointed.
Input: A location $l \in P$ at which to serialize $x$.
procedure SCR($P, x, \tau, l$)
    call CompileTimeProcess($\tau$)
    at $l$ in $P$ emit “clear memoizeSet; srlz_$\tau$(x);”
procedure CompileTimeProcess($\tau$)
    if $\tau$ = int ∨ $\tau$ = float ∨ $\tau$ = enum then
        emit “srlz_$\tau$($\tau$ &x) { write x to file; }”
    else if $\tau = \tau'$ ptr then
        call CompileTimeProcess($\tau'$)
        emit “srlz_$\tau$($\tau$ &x) {
    write x to file; if (x $\notin$ memoizeSet) {
    add x to memoizeSet; seralize_$\tau'$(*x); } }”
    else if $\tau = \tau'$ array[$n$] then
        call CompileTimeProcess($\tau'$)
        emit “srlz_$\tau$($\tau$ &x) { ”
        for $i \in 1 \dots n$ do
            emit “ srlz_$\tau'$(x[$i$]);”
        emit “}”
    else if $\tau = \{f_1 : \tau_1, \dots, f_n : \tau_n\}$ struct then
        for $i \in 1 \dots n$ do
            call CompileTimeProcess($\tau_i$)
        emit “srlz_$\tau$($\tau$ &x) { ”
        for $i \in 1 \dots n$ do
            emit “ srlz_$\tau_i$(x.f$_i$);”
        emit “}”
    else if $\tau = \{f_1 : \tau_1, \dots, f_n : \tau_n\}$ union then
        $i = \arg\max_{x \in 1 \dots n}$ sizeof($\tau_x$)
        call CompileTimeProcess($\tau_i$)
        emit “srlz_$\tau$($\tau$ &x) { srlz_$\tau_i$(x.f$_i$); }”
```

τ. For some primitive scalar types (e.g., int, float, enum), SCR emits a procedure that directly serializes the value to a file. For pointers, we emit a procedure that, at run-time, first serializes the pointer address and then checks to see if that address has been previously processed during this run-time serialization using a memoization set. This serves to detect cycles, such as in a circularly linked list. If the address has not been memoized, it is added to the memoization set, and the serialization function for the dereferenced value is called. Note that the address is serialized in both cases; this allows the deserialization code, which also maintains a memoization set, to operate in lock-step. For arrays, SCR emits code that iteratively serializes every array element. Similarly, structures are handled by serializing every field. For union types, we determine the type of the largest field at compile time, then recursively emit serialization code for that type. Note that union types, which normally complicate source-level analyses as they represent implicit casts between values that share the same storage space, are easier for SCR to handle. We need not know or track which variant field of the union was accessed last or will be accessed next since we serialize the entire storage space regardless.

Finally, after all of the relevant serialization functions have been generated, we emit a call to serialize x at the user-specified location

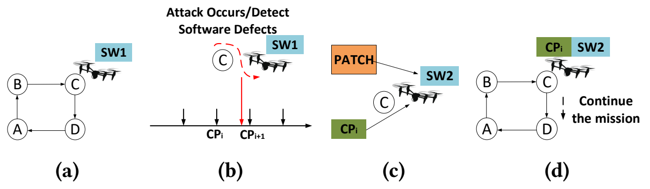

Figure 1: An indicative checkpoint-detect-repair-restore use case for SCR, detailed in Section 4.2.

l in P. The result of the algorithm is the original program integrated with the emitted mutually-recursive serialization functions and initial serialization call. For simplicity, we present the algorithm for a single annotated variable x; the extension to multiple variables is direct. The generation of deserialization code is symmetric.

3.2 Annotations

At the coarsest level, the user provides annotations to SCR in the form of the set of source-level variable names x to checkpoint and the source-level location l at which to call the serialization and deserialization functions (Algorithm 1). By default, SCR checkpoints all fields and types transitively reachable from each variable x. However, in an AVS setting, not all fields or types contain critical data that should be checkpointed. For example, some fields may hold real-time sensor data which should be re-acquired on restoration rather than restored. Similarly, some fields may be tainted by an attacker and should be recalculated rather than restored. To address this issue, SCR allows the user to specifically include or exclude fields or types from checkpointing. Because the serialization functions generated are symbolic, entire contiguous data structures need not be stored: fields or types can be excluded from both serialization and deserialization. In our prototype, fields and types are specified through simple `#pragma` or comment annotations.

4 SYSTEM & EXPERIMENTAL DESIGN

In this section, we first introduce the specimen system we used to evaluate SCR, our proposed selective checkpoint and restore algorithm. We then discuss an indicative use case to demonstrate how SCR can apply to a modern autonomous vehicle software system. Finally, we detail our experimental design.

4.1 Specimen System

We evaluate SCR on ArduPilot, an open-source autopilot system atop a Unix-like operating system with a real-time kernel. Autonomous vehicle systems are particularly relevant to checkpoint and restore systems [16, 17, 42, 64] but also strongly benefit from patching and other resiliency actions [13, 62] (see Section 2.2). Our experiments use the ArduCopter module of ArduPilot to control a quadcopter autonomous aerial vehicle. The vehicle is controlled remotely via the Micro Autonomous Vehicle Link (MAVLink) protocol. This high-level architecture is indicative of popular autonomous vehicles like the Erle-Copter and various Raspberry Pi-based copters.

4.2 Indicative Use Case

We consider a use case in which an uncrewed autonomous vehicle (UAV) completes a four-waypoint mission during which undesirable behavior is observed and repaired, such as depicted in Figure 1.

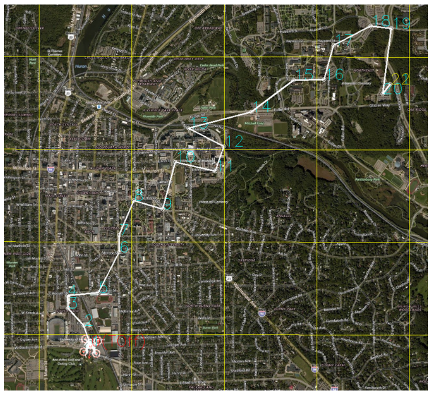

Figure 2: `BusRoute`: a 21-waypoint UAV mission plan of a bus route in the area of Ann Arbor, Michigan, USA.

Prior to deployment, the developer annotates relevant data structures, executes SCR, and loads the modified binary onto the vehicle. During mission execution (1a), the selected state is repeatedly checkpointed. Near waypoint C, the operator or vehicle detects an attack or anomaly (e.g., via various off-the-shelf techniques [5, 47]), which necessitates patching the software to complete the mission (1b). The automated repair system [9, 31] constructs a patch while the vehicle loiters safely [13]. The patch is deployed, potentially changing certain aspects of the code and data layout of the system. The latest checkpoint, captured with respect to the pre-patch software, is then restored into the post-patch system (1c). Finally, the mission resumes and runs to completion (1d).

Patches to UAV software may introduce new—or reorder existing—fields within a structure. Because SCR restores variables based on source-level names rather than original memory locations, our system can successfully resume the software after such a patch is applied. In contrast, traditional approaches may fail due to the alterations in memory layout.

4.3 Experimental Design

We designed our experiments to measure SCR's ability to successfully checkpoint and restore in the face of patches, as well as its performance overhead. We compare it to CRIU [35], a state-of-the-art baseline (see Section 2). To admit reproducible experimentation for both tools, we use the Software in the Loop (SITL) simulator provided by ArduPilot, but we have validated the Red Team-provided arc injection bug and APR patch in live flight (see Section 4.3.4).

4.3.1 Variables to Checkpoint. Our approach requires manual annotation of mission-critical variables (see Sections 2.4 and 3.2). After manual analysis of the source code for mission-critical variables, we selected all of the primitive fields of the `Copter` class as well as its `AP_Mission` and `AP_AHRS` object fields. We evaluate the difficulty of this analysis and annotation effort in Section 5.2.

4.3.2 Missions. We applied SCR and CRIU to two long-running, autonomous missions: `ISTAR` and `BusRoute`. The `ISTAR` mission is taken from previously-published work also using ArduPilot software [13]. This mission is an indicative simplification of a certain class of intelligence, surveillance, target acquisition, and reconnaissance (ISTAR) activities deemed relevant to the US Air Force. This mission contains six waypoints around an airfield. The entire mission route is 0.13 kilometers. Our second mission (`BusRoute`) follows a municipal bus route in Ann Arbor, Michigan, as shown in Figure 2. This mission contains 21 waypoints, each one of which corresponds to a bus stop. The maximum diameter of this mission is 4.72 kilometers. The entire mission route is 7.55 kilometers.

4.3.3 Control Software. Our use case includes checkpointing the state associated with one version of the software, modifying the software, and then restoring the state into the new version of the software (see Section 4.2). For our experiments, we considered both human-written and APR-generated patches: (a) we collected five different versions of the ArduPilot software from its GitHub commit history. The differences between software versions correspond to patches (e.g., to fix a defect or security vulnerability); (b) we used a version of ArudPilot seeded with an arc injection security vulnerability by a Red Team paired with its associated patch (produced by Darjeeling, an APR algorithm [58]). This was taken from part of a US Air Force assessment, in which Assured Information Security highlighted attacks, both found and seeded, against ArduPilot.

The five versions in (a) are the first five consecutive commits starting from Jan 23, 2018 that contain source code changes related to ArduPilot's ArduCopter module (i.e., not configuration file changes or changes to unused modules). They are indicative patches made by human developers. Each patch modifies between 60 and 160 lines of code. We refer to these five versions as $V0$ through $V4$. The arc injection vulnerability in (b) was seeded in $V0$.

4.3.4 Experiment setup. We designed our experiments following the steps below to simulate a scenario in which the UAV encounters a bug during a mission, is patched, and resumes its mission. For the human-written patches (a), we: (1) Preselect a random point in the mission to correspond to the simulated defect; (2) Start the ArduPilot quadcopter and its associated support software; (3) Load and start the mission; (4) When the preselected point is reached, checkpoint the system state and terminate the control software; (5) Resume a different version of the ArduPilot system (i.e., apply a patch); (6) Restore checkpointed data; and (7) Complete the mission.

For the APR-generated patch (b), instead of simulating a defect at a random point, we used the exploit provided by the Red Team, which used specially-crafted MAVLink packets to cause a system crash. During the mission, we checkpointed the state repeatedly and used the latest checkpointed state to resume the system with an APR-generated patch applied that fixes the security vulnerability.

In all cases, we repeated these steps 50 times for every version using both SCR and CRIU. We also repeated the missions 50 times without any checkpointing or restore operations to establish a baseline for overhead comparisons.

4.3.5 Measurement. We measured the time and space overhead as well as whether or not each mission completed.

CRIU. CRIU time overhead includes two parts: (1) The time of the mission from waypoint 0 to the preselected checkpoint location. This time period includes the time of CRIU checkpointing. (2) The time from invoking CRIU to restore (i.e., resume the ArduPilot application and mission) to reaching the last waypoint of the mission. This period includes the time taken by CRIU to restore the system.

SCR. SCR time overhead includes three parts: (1) the time of the mission from waypoint 0 to the preselected checkpoint location or the location where attack happens. This time period includes checkpointing time. (2) The launch time of ArduPilot and its associated software when resuming a new version of the system. (3) The time from calling the restore function to reaching the last waypoint. This period includes the time to selectively restore mission state.

The difference between the measures is due to the different architectures between CRIU and SCR. Because CRIU is implemented atop the `/proc` file system and the process tree, its restore action does not require a re-launch or re-initialization of ArduPilot or its associated software. In contrast, SCR operates symbolically on in-memory variables, and restoration proceeds by executing the new binary until it reaches the restoration point. Note that in our approach, the software system is started twice: once at the beginning of the mission and once before the restore. By contrast, with CRIU, the system is only started once: CRIU stores all of the state and does not need to rerun any start up code. The cost of starting the software twice is included in our evaluation times.

In addition to the time overhead, we also measured the space overhead for every experiment: the memory space necessary to store all the checkpointing information (i.e., on disk).

5 EXPERIMENTAL RESULTS

We address the following research questions:

RQ1 How does SCR's overhead compare to baselines?
RQ2 Does SCR successfully restore state after patching?
RQ3 How hard is it for programmers to annotate for SCR?

5.1 Restoration Success and Performance

Table 1 shows the restoration success and overhead for two UAV missions (`ISTAR` and `BusRoute`) using our the proposed selective checkpoint and restore algorithm (SCR) and CRIU (see Section 4.3.4). Different rows correspond to different patches (i.e., software version changes) applied between checkpointing and restoration. We also list the baseline performance of running each mission without any checkpointing, patching or restoration. Table 1 also shows the restoration success and overhead with the Red Team provided ArduPilot version and an APR-generated patch for the `ISTAR` mission using SCR and CRIU. The Red Team attack uses the `ISTAR` waypoints but runs the mission at a slower speed. The ± values show one standard deviation over 50 repeated measurements.

Time overhead. The UAV system with SCR took longer than the CRIU system to complete each mission. As noted in Section 4.3.5, SCR requires restarting the ArduPilot software to support mid-mission patching. The average launch time of ArduPilot for our experiments is 1.96 seconds. This restart time is a fixed cost that does not vary with mission complexity; it is more relevant in the shorter `ISTAR` mission than the longer `BusRoute` mission.

If the ArduPilot re-launch time is removed, no statistically significant difference remains between the CRIU and SCR time overheads. Both CRIU and SCR introduced a time overhead of 0.2–5.5 seconds, compared to not checkpointing (i.e., the Baseline), over all human-patched experiments. The majority of this overhead corresponds to caching and other indirect effects. SCR introduced a time overhead of 9 seconds approximately in the APR-patched experiments. This time overhead is larger because these experiments were conducted at a lower simulation speed—the Red Team exploits depend on real time. SCR storage times were quite low relatively (e.g., it required an average of 0.18 ms for checkpointing and 0.14 ms for restoration) and are thus not reported separately.

Table 1: Restoration success and overhead measurements for SCR and CRIU on the `ISTAR` and `BusRoute` missions using both human-written and APR-generated patches. The "Baseline" row corresponds to running the mission without checkpointing, patching or restoration. Cells marked ✗ could not successfully execute, resulting in mission failure.

`ISTAR` Mission, Human Patches

	Patch	Time (s)	Space	Success
Baseline	none ($V0$)	7.45 ± 0.26	n/a	n/a
CRIU	$V0 \rightarrow V0$	8.15 ± 0.55	822.37 ± 19.96KB	100%
	$V0 \rightarrow V1$	8.18 ± 0.55	823.44 ± 4.23KB	100%
	$V0 \rightarrow V2$	✗	✗	✗
	$V0 \rightarrow V3$	✗	✗	✗
	$V0 \rightarrow V4$	✗	✗	✗
SCR	$V0 \rightarrow V0$	9.92 ± 0.86	290B	100%
	$V0 \rightarrow V1$	9.73 ± 0.37	290B	100%
	$V0 \rightarrow V2$	9.83 ± 0.38	290B	100%
	$V0 \rightarrow V3$	9.80 ± 0.69	290B	100%
	$V0 \rightarrow V4$	9.76 ± 0.92	290B	100%

`BusRoute` Mission, Human Patches

	Patch	Time (s)	Space	Success
Baseline	none ($V0$)	250.83 ± 0.11	n/a	n/a
CRIU	$V0 \rightarrow V0$	251.03 ± 0.20	821.52 ± 2.87KB	100%
	$V0 \rightarrow V1$	254.31 ± 21.09	823.60 ± 15.18KB	100%
	$V0 \rightarrow V2$	✗	✗	✗
	$V0 \rightarrow V3$	✗	✗	✗
	$V0 \rightarrow V4$	✗	✗	✗
SCR	$V0 \rightarrow V0$	254.59 ± 7.21	290B	100%
	$V0 \rightarrow V1$	253.94 ± 0.87	290B	100%
	$V0 \rightarrow V2$	254.02 ± 0.79	290B	100%
	$V0 \rightarrow V3$	256.48 ± 14.17	290B	100%
	$V0 \rightarrow V4$	255.46 ± 11.65	290B	100%

`ISTAR` Mission, Red Team Bug, APR Patch

	Time (s)	Space	Success
Baseline	47.46 ± 0.16	n/a	n/a
CRIU	✗	✗	✗
SCR	56.60 ± 1.53	290B	100%

Space overhead. For these missions, SCR, which checkpointed only the mission- and security-critical data implicated by the user annotations, required only 290 bytes. By contrast, CRIU, which does not require annotations but does save the majority of the process state, required 822KB. In these experiments, SCR demonstrates a 2000× reduction in storage required by checkpointing. We note that both CRIU and SCR captured enough state for the mission to progress: neither had to restart the mission "from scratch" or revisit any waypoints after being interrupted and resumed. SCR adds 12KB to the ArduPilot binary, which is less than 1% of the total size of ArduPilot (1.57Mb). CRIU requires 1.6Mb of disk space.

Restoration success. Both CRIU and SCR can successfully continue the mission when the checkpointed and resumed software are identical ($V0 \rightarrow V0$ in Table 1). However, when the system was resumed with a patched version of the software, SCR had a 100% success rate but CRIU failed over 75% of the scenarios. In particular, CRIU only succeeded at restoring after the small patch that did not affect the actual code part of ArduPilot ($V0 \rightarrow V1$); CRIU failed to restore for $V0 \rightarrow V2$, $V0 \rightarrow V3$, $V0 \rightarrow V4$, or when ArduPilot is patched via APR. In fact, the APR patch that fixes the Red Team security bug only changes a single line of code, but CRIU was unable to resume after it was applied. An ✗ denotes situations in which CRIU failed to resume a new version.

CRIU restoration errors manifested as segmentation faults or other crashes induced by memory errors. Since CRIU saves memory exactly "as is", patches that changed how the software expected memory to be laid out (e.g., reordering, adding, or removing fields from critical data structures) led to memory errors in practice. By contrast, since SCR restores variables and fields symbolically, a patch that adds a new field Z between old fields A and B will result in SCR restoring the checkpointed values of A and B but using the value of Z as initialized when the system is re-launched. This does not guarantee restoration, but worked for all scenarios considered.

5.2 Annotation Effort

SCR requires the user or developer to annotate mission- or security-critical data for checkpointing (see Section 3). This is done statically, at the level of source-level program symbols (i.e., variable or field names), rather than dynamically, in terms of pointers or addresses (cf. libckpt [38]). While particular fields or types can be individually included or excluded, the most common case is to specify a variable, at which point SCR will recursively traverse type information to generate checkpointing and restoration code for all information reachable from that variable (via fields or pointers).

For these experiments, one author was familiar with the ArduPilot project and identified mission-critical data for checkpointing and restoration, including information such as the current mission item being executed and the arming status of the vehicle's motors. While the $V0$ version of ArduPilot and associated libraries used in these experiments contain over 2,275,000 lines of code in total, we found that we only needed to inspect 1,963 lines—less than one tenth of one percent of the code—to annotate 96 variables relevant to the previously-identified critical state. Our annotation effort was focused on three structures in the source code: `Copter`, `AP_AHRS` and `AP_Mission`, which store data related to high-level event loop, positional awareness, and mission state information.

The annotation burden with SCR is minimized because we leverage the existing *domain expertise* of developers and users. Studies of code ownership and the role of knowledge in software development suggest that experienced developers are often familiar with the key data, structures, and design in their software [3, 43, 46]. We abstract away the inner workings of our Checkpoint/Restore mechanism from the required annotations. This corresponds to a lighter annotation burden than other projects such as ESC/Java [8] or Liquid Types [48], where annotations can often include difficult-to-write invariants and pre- and post-conditions. SCR annotations more closely align with pickling combinators [20] or library-level serialization interfaces, a familiar interaction to users.

5.3 Experiment Conclusions

In our experiments, SCR's time overhead was comparable to the state-of-the-art CRIU technique (slower only by the time required to restart the software after the patch, and not significantly different otherwise). SCR's space overhead was significantly lower than CRIU's, requiring 290 bytes rather than 820 kilobytes. Most importantly, SCR was able to successfully resume missions after deployment of developer- or APR-written patches in 100% of examined cases, while CRIU only succeeded 36% of the time (i.e., for the simplest patches). SCR requires manual annotations, but less than one tenth of one-percent of the program had to be annotated.

6 THREATS TO VALIDITY

We discuss several threats to validity. First, we only evaluated SCR on one open-source project. Although this AVS software provides straightforward measurements, the results may not generalize to all types of software systems. However, ArduPilot is a large, diverse software ecosystem in an increasingly important area of research. Second, the annotation burden may vary for different applications. While ArduPilot is an indicative software system, we do make the assumption that type-guided annotations can be succinct. This assumption would fail for systems with crosscutting concerns, as in aspect-oriented programming [21] or for certain functional programming languages. Third, although SCR allows the system to resume after patches, unlike traditional Checkpoint/Restore approaches, neither approach can handle certain vulnerabilities, such as information leaks. Finally, SCR, CRIU and similar systems typically require exclusive access to the data being checkpointed or restored, which is complicated by threading.

7 CONCLUSIONS

Software systems such as autonomous vehicles face challenges from software defects, environmental changes, and security attacks. Failure transparency is one solution that can address software defects or security attacks that result in system failures. This paper presents a selective checkpoint and restore algorithm that uses symbolic, type-guided annotations to save only mission-critical system state in an effort to provide failure transparency in software systems like AVS. Unlike traditional Checkpoint/Restore solutions, our approach allows the system to resume from a different version of the software, so that necessary system repairs (e.g., generated an automated program repair technique) can be applied to fix software bugs between checkpoints and restorations. We evaluated our algorithm with an open source AVS software package. Compared to a state-of-the-art baseline, our approach had comparable time overhead, orders of magnitude lower space overhead, and was able to restore after complicated patches, providing failure transparency and system resiliency in autonomous vehicle software.

REFERENCES

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[2] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[3] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Foundations of Software Engineering*, pages 4–14, New York, NY, USA, 2011. ACM.

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *ACM Sigplan Notices*, volume 38, pages 84–94. ACM, 2003.
[5] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.
[6] Y. Chen, J. S. Plank, and K. Li. Clip: A checkpointing tool for message-passing parallel programs. In *Supercomputing*, pages 1–11, 1997.
[7] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39, 2014.
[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*, pages 500–517, 2001.
[9] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: a survey. In *International Conference on Software Engineering*, pages 12–19, 2018.
[10] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *ACM SIGPLAN Notices*, volume 42, pages 321–336. ACM, 2007.
[11] E. Hendriks. Bproc: The beowulf distributed process space. In *Supercomputing*, pages 129–136, 2002.
[12] M. Hicks and S. Nettles. Dynamic software updating. *Trans. Programming Languages and Systems*, 27(6):1049–1096, 2005.
[13] K. Highnam, K. Angstadt, K. Leach, W. Weimer, A. Paulos, and P. Hurley. An uncrewed aerial vehicle attack scenario and trustworthy repair architecture. In *Dependable Systems and Networks*, pages 222–225, 2016.
[14] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy*, pages 191–205, 2013.
[15] D. Jackson. Alloy: a lightweight object modelling notation. *Trans. Software Engineering and Methodology*, 11(2):256–290, 2002.
[16] A. Y. Javaid. *Cyber security threat analysis and attack simulation for unmanned aerial vehicle network*. PhD thesis, University of Toledo, 2015.
[17] A. Y. Javaid, W. Sun, V. K. Devabhaktuni, and M. Alam. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *Technologies for Homeland Security*, pages 585–590, 2012.
[18] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *Trans. Parallel and Distributed Systems*, 10(6):560–579, 1999.
[19] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Computer and Communications Security*, pages 272–280, 2003.
[20] A. J. Kennedy. Functional pearl pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.
[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.
[22] A. Kim, B. Wampler, J. Goppert, I. Hwang, and H. Aldridge. Cyber attack vulnerabilities analysis for unmanned aerial vehicles. In *Infotech@ Aerospace*, pages 1–30, 2012.
[23] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *International Static Analysis Symposium*, pages 218–234. Springer, 2005.
[24] O. Laadan and S. E. Hallyn. Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium*, volume 159, 2010.
[25] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.
[26] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Notices*, 51(1):298–312, 2016.
[27] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Operating System Design and Implementation*, page 20, 2000.
[28] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.
[29] J. A. Marty. Vulnerability analysis of the mavlink protocol for command and control of unmanned aircraft. Technical report, Air Force Institute of Technology, 2013.
[30] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, pages 691–701, 2016.
[31] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.
[32] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. *Performance Evaluation Review*, 34(1):216–227, 2006.
[33] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering*, pages 772–781, 2013.
[34] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.
[35] Pavel Emelyanov. Checkpoint/restore in userspace. In *https://criu.org*, 2012.
[36] L. Perkov, N. Pavković, and J. Petrović. High-availability using open source software. In *Information and Communication Technology, Electronics and Microelectronics*, pages 167–170, 2011.
[37] E. Pinheiro. Epckpt: Eduardo pinheiro checkpoint project, 2004.
[38] J. S. Plank, M. Beck, G. Kingsley, and K. Li. *Libckpt: Transparent Checkpointing under Unix*. January 1995.
[39] M. L. Powell and B. P. Miller. Process migration in demos/mp. In *Operating Systems Review*, volume 17, 1983.
[40] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance*, pages 180–189, 2013.
[41] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36, 2015.
[42] High-assurance cyber military systems (HACMS). https://www.darpa.mil/program/high-assurance-cyber-military-systems, 2015.
[43] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *International Conference on Software Engineering*, pages 491–500, 2011.
[44] R. Ramey. boost c++ libraries, 2004.
[45] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Code Generation and Optimization*, pages 243–254, 2005.
[46] P. N. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, Jan 1999.
[47] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.
[48] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Sigplan Notices*, volume 45, pages 131–144, 2010.
[49] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Parallel and Distributed Processing Symposium*, 2005.
[50] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *J. High Performance Computing Applications*, 19(4):479–493, 2005.
[51] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *Trans. Computer Systems*, 2(2):145–154, 1984.
[52] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.
[53] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.
[54] D. P. Shepard, J. A. Bhatti, T. E. Humphreys, and A. A. Fansler. Evaluation of smart grid and civilian uav vulnerability to gps spoofing attacks. In *Radionavigation Laboratory Conference Proceedings*, 2012.
[55] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.
[56] V. L. Thing and J. Wu. Autonomous vehicle security: A taxonomy of attacks and defences. In *Cyber, Physical and Social Computing*, pages 164–170, 2016.
[57] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering*, pages 303–314, 2018.
[58] C. Timperley and C. Le Goues. Darjeeling: a language-agnostic search-based program repair tool. In *https://github.com/squaresLab/Darjeeling*, 2020.
[59] R. van Tonder and C. Le Goues. Static automated program repair for heap properties. In *International Conference on Software Engineering*, pages 151–162, 2018.
[60] Vmadump. https://bproc.sourceforge.net, 2002.
[61] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Fault-Tolerant Computing*, pages 00–22, 1995.
[62] W. Weimer, S. Forrest, M. Kim, C. Le Goues, and P. Hurley. Trusted software repair for system resiliency. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pages 238–241, 2016.
[63] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.
[64] A. M. Wyglinski, X. Huang, T. Padir, L. Lai, T. R. Eisenbarth, and K. Venkatasubramanian. Security of autonomous systems employing embedded computing and sensors. *IEEE Micro*, 33(1):80–86, 2013.
[65] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.