Research Statement

Dr. Luís Gabriel Ganchinho de Pina

Modern software is plagued with errors that compromise its availability and integrity; causing crashes, exposing sensitive data, and allowing software to behave maliciously. Despite a large body of work in finding and fixing these errors, there is a gap between having a fix available and actually deploying it in production. This enables attacks on high-profile targets using known errors, such as the recent Equifax incident.

Why does this happen? I believe developers lack the tools to develop updates as a first-class feature, and operators face obstacles to deploy updates in production. My goal is to provide both developers and operators with tools and techniques that bridge the update gap in practical deployments of real applications.

The main obstacles operators face when deploying updates are update-related disruption and loss of state, and uncertainty about the reliability of the updated system. My research in **Dynamic Software Updating (DSU) [5, 8]** minimizes the disruption and eliminates any data loss. My research in **Multi-Version Execution (MVE) [6, 3], combined with DSU [4]**, allows operators to run the updated version concurrently with the old version, which allows to detect failed updates and revert them without losing any data.

State-of-the-art research on DSU requires developers to specify *when* a program can be updated, and *how* to transform its state to be compatible with the new version. My research on DSU automates that effort as much as possible **[8]**, and provides the ability for **Systematic DSU Testing [7]**. Unfortunately, debugging and analyzing updates remains an unsolved challenge, especially for realistic systems. My work on **Lightweight Checkpoint/Rollback Support [1]** can be extended to allow time-travel debugging, retrying the update many times in sequence as needed without polluting the system state. My work-in-progress on **Concolic Execution for Managed Languages** uses binary instrumentation to explore the program state-space by controlling inputs, which can be extended to find errors in updates in short order and at low cost, before the update is made available.

I believe that a pragmatic approach to reliable software requires a combination of: (1) a lightweight dynamic analysis that detects when the program behaves incorrectly, and (2) a powerful offline analysis that finds errors before new versions are made available. My vision for bridging the update gap further adds the ability to deploy updates online as soon as they are made available. Looking farther ahead, natural synergies between online and offline analyses further improve the reliability of software (*e.g.,* guiding the offline analysis with insights generated online, or using the offline results to choose what to monitor online).

Research Experience

Dynamic Software Updating (DSU) *OOPSLA'14, ICST'15*

Typically, to update a running process, one needs to stop it first and then restart it in the next version. This causes a period of zero service and loss of important program state that cannot be persisted (*e.g.,* active network connections, contents of main memory). Dynamic Software Updating (DSU) solves both problems by updating a process in-place, which minimizes the pause in service and preserves all program state by transforming it to an equivalent version that is compatible with the new code. In my research, I explore how to apply DSU to novel programming models **[5]** and I improve the state-of-the-art for managed runtimes by showing how to perform DSU on stock VMs and how to migrate the program state lazily **[8]**.

DSU, unfortunately, is not a panacea. Developers must support updates as another application feature, with the risk of introducing update-related errors. My research addresses this problem by exploring how to test updates **[7]**. Furthermore, I combine DSU with MVE **[4]** to keep update failures from crashing the whole system by forking the outdated process, starting an MVE execution, and performing the update in the background process. A failed update can be detected by comparing with the outdated version, and can be discarded by simply terminating the updated version; all while the outdated version provides full availability.

Multi-Version Execution (MVE) ***USENIX ATC'17, ACM CF'18***

Multi-Version Execution (MVE) executes many versions of the same program at the same time. MVE can increase the availability of the underlying system by deploying versions with different failure models (*e.g.,* different releases of the same program). When a fault happens, MVE allows execution to continue with all the surviving versions. MVE can also be used to detect attacks, and stop them, by running all versions in lock-step and stopping when versions disagree on the next action to take. Each version executes in a separate process, and the MVE runtime synchronizes all executions, for instance by ensuring that all processes issue the same (or a similar) sequence of system calls. When reading data from disk/network, the MVE system reads the data first, buffers it, and then provides that same data to each process.

MVE allows to deploy versions of the same program with different failure modes (*e.g.,* stack growing in different directions in each version) to either increase availability, continuing execution with the versions that survive a fault; or detect errors/exploits, stopping when it detects divergences in the processes under MVE. My research shows how to use efficient MVE support to monitor a program on separate processes **[3]**, with three main advantages: (1) deploy out-of-the-box incompatible dynamic analyses in the same execution (*e.g.,* address/memory compiler sanitizers with Valgrind), (2) fully mask the overhead of heavyweight analysis for common application scenarios (*i.e.* Valgrind with native performance), and (3) trade-off more machines for native latency to enable deploying heavyweight analyses in a distributed scenario. My research also shows that it is feasible to deploy versions with known benign divergences **[6]**, for instance different versions of the same program or using different configurations (*e.g.,* logging enabled/disabled).

For my research in MVE, I led a team of graduate and undergraduate students that resulted in publications in top-tier conferences, and laid the groundwork that explored the feasibility of a grant (later funded [2]).

Lightweight Checkpoint/Rollback Support ***ECOOP'18***

Checkpoint/rollback (C/R) allows developers to save the program state at any point in the program, perform some speculative changes, and later revert back to that state if needed. Checkpoint/rollback is a building block for sophisticated development tools (*e.g.,* time-travelling debuggers) and program analyses (*e.g.,* fuzz-testing of stateful programs). I have implemented a state-of-the-art C/R tool for Java **[1]** that runs on unmodified out-of-the-shelf JVMs (*e.g.,* Oracle's HotSpot), and that saves/restores the program state efficiently as it is accessed for the first time after a checkpoint/rollback; all with a trivial 5% performance penalty.

Ongoing and Future Work

I am currently working on offline analysis of managed languages through concolic execution, which will enable my research goals by using the same technique to bridge the update gap. I am also working on other techniques to improve the reliability of software, such as Intrusion Detection Systems, combining it with my previous research. Besides my current work, I have concrete plans for next projects to reach my research goals.

Guided Offline Analysis through Concolic Execution. *Symbolic execution (SE)*, as opposed to concrete execution, allows to explore the state-space of a program by declaring symbolic variables that can take any value, and computing how the program constraints them as it executes. For instance, considering variable `i` is is symbolic, the statement `if (i<10)` causes SE to consider both cases of the condition, adding `i<10` to the list of current constraints (path condition) when the conditional is taken, and `i>=10` otherwise. Then, SE sends both sets of constraints to a SAT/SMT solver and, if both are feasible (given previous constraints captured in the program), SE attempts to explore both paths of this program. SE can thus discover paths that lead to errors (*e.g.,* failing assertions) and generate inputs that exercise those paths by sending that path condition to a solver. Although powerful, SE scales poorly: It generates a large number of feasible paths, and

choosing which to explore is hard; and it models uninteresting but important portions of the program (*e.g.,* standard libraries). Furthermore, SE support for managed languages lags behind compiled languages.

Concolic execution (CONCrete + symbOLIC) provides a viable solution to the scalability problem: Instrument the code to capture constraints (as SE) but run that program on a totally concrete input. Then, capture the path condition of that concrete input, and explore new paths by manipulating constraints (*e.g.,* negating a subset to explore a different path).

I am applying concolic execution to Java programs by using a taint-tracker to compute constraints on interesting variables (*e.g.,* data flowing through a server that is controlled by the client). My early results demonstrate the improved scalability as it works with the Tomcat application server, which relies on features that analyses often struggle to process, such as multi-threading, reflection, complex class-loading, and code-generation. Furthermore, early results show that my approach generates more and better paths than the state-of-the-art: When guiding the search towards slow paths to find algorithmic complexity attacks, my approach finds better (slower) inputs than the state-of-the-art, taking less time to do so. This approach is the basis of a grant to be submitted to the NSF Secure and Trustworthy Cyberspace (SaTC) in the Fall 2018.

This direction of research can be extended in several ways. For instance, combining it with lightweight checkpoint/rollback support **[1]** eliminates false positives, in which the analysis finds an error simply due to program state polluted by previous executions. Another direction is to apply concolic execution to DSU, exploring the interaction between: old code, new code, program-state transformation logic, and the set of program points where updates are allowed. Yet another direction is to leverage the constraints captured to reduce the manual effort needed for developers to support updates, for instance by generating state transformation code automatically.

Intrusion Detection Systems (IDSes) for Managed Languages. IDSes have a track-record of detecting attacks early and accurately. However, applying an IDS to a managed language (*e.g.,* Java, Javascript) is not trivial, as the internals of the VM can mask legitimate executions from attacks. MVE for managed languages suffers from the same problem, as it needs to ignore the internals of the VM but still synchronize the observable behavior of the program being executed. Solving one problem provides a solution to the other. Furthermore, IDS models are expensive to build, typically requiring specialist-written rules or many hours of automated training. Once a system is updated, the IDS model needs to be updated as well. Combining MVE with IDS allows to automate and speed-up the model update, even deploying the new version as soon as possible and creating the new model on-the-fly, using the old model/program as an oracle. I worked with Radmin, an IDS based on probability-finite-automata (PFA) that provides state-of-the-art detection accuracy for resource consumption attacks. Radmin provided me with the opportunity to liase with a company using a research prototype (Vencore), and what it takes for a prototype to go from research into industry.

Scaling Program Analyses with Crowdsourced Execution Data. Online analysis can work on actual inputs that the program receives. Lightweight techniques make this feasible for production systems; which process real inputs instead of realistic test cases. Offline analysis can detect latent errors in existing programs, even if that fault has not been ever observed before. Therefore, there is a huge potential to combine offline analysis with online monitoring, for instance to guide the analysis towards execution traces observed online; and with DSU, for instance to ensure that an update will not crash or introduce new errors when applied.

References

[1] Jonathan Bell and Luís Pina. CROCHET: Checkpoint and rollback via lightweight heap traversal on stock JVMs. In *ECOOP*, 2018.

[2] Cristian Cadar and Alastair Donaldson. Automatically detecting and surviving exploitable compiler bugs. 2018. http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/R011605/1.

[3] Luís Pina, Anastasios Andronidis, and Cristian Cadar. FreeDA: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *ACM Computing Frontiers (CF)*, 2018.

[4] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. MVEDSUA: Higher availability dynamic software updates via multi-version execution. (under submission).

[5] Luís Pina and João Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proceedings of the Fourth Workshop on Hot Topics in Software Upgrades (HotSWUp 2012)*, June 2012.

[6] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A DSL approach to reconcile equivalent divergent program executions. In *USENIX ATC*, 2017.

[7] Luís Pina and Michael Hicks. Tedsuto: A general framework for testing dynamic software updates. In *ICST*, 2016.

[8] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA*, 2014.