Alternative Analysis: Implementing Agile Under-Constrained Symbolic Execution in Software Verification

GMU
Dylan Knoff
4/1/2025

Problem Statement

Software engineers of the modern day are expected to be cross-functional individuals with increasing pressure to deliver secure, reliable code at a consistent and constant pace that aligns with Agile workflows. When it comes to developing secure software and bug hunting, traditional fuzzing techniques, while often effective at scale, lack the determinism of their symbolic alternatives and suffer from highly limited capabilities when it comes to code coverage. Instrumented fuzzing relies on mutating inputs, while the codebase and reachability of certain execution paths are not highly considered in this testing technique. Symbolic execution tools utilize the binary being tested to develop SMT theorems from program branches and effectively providing more comprehensive and "intelligent" tests. If a software engineer wanted to see what input was needed to reach a certain execution path in their product, they could reliably figure out that exact input using symbolic execution. Most symbolic execution engines refer to a process known as "concolic analysis", with "concolic" referring to a combination of both concrete and symbolic variables being used in analysis. However, while concolic analysis provides the precision in testing that fuzzing lacks, it suffers from a scalability issue. This issue is due to "path explosion", which can arise from program blobs that cause an exponential number of potential execution paths. A promising middle ground is under-constrained symbolic execution (UC-SE), a variation that allows starting execution at an arbitrary point in the target program rather than the entry point, enabling more modular analysis. However, the integration of UC-SE into production software packages requires several considerations to overcome the challenges of starting accurate analyses midway through a program. These considerations include context awareness, precision, scalability, and overhead. This alternatives analysis aims to identify the most effective approach to implementing UC-SE in a way that aligns with Agile principles and common engineering workflows.

Alternatives Overview

In this alternatives overview I will convey four possible alternatives which can address key issues, meet evaluation criteria, and overall enhance an engineer's ability to accurately test products using under-constrained symbolic execution. The first of these alternatives is considering precomputed initial states. This would initialize under-constrained analysis with precomputed memory and register values based on common execution contexts. This simulates more realistic and predictive program behavior while also reducing the size of the symbolic state. Furthermore, this alternative is frequently

preferred by engineers as it allows explicit symbolization, consider how an input file might be stored in memory.

The second alternative would be that of purposeful concretization. Meaningful concretization, when appropriate, can improve precision by using set, concrete values rather than symbolic values. One of the inherent pitfalls of UC-SE is that it assumes all input and uninitialized memory states are symbolic. By selectively assigning concrete values to certain inputs and memory regions, the analysis becomes more focused and avoids exploring meaningless paths. It ensures that the software implementing SE will prioritize paths that are more likely to be practically encountered while limiting the initial state of the analysis engine.

Third is priority heuristics. Using priority heuristics to drive path exploration towards high-risk execution paths would also be a valuable alternative in discovering high-value and high-likelihood bugs first. This would involve intelligently selecting the most relevant execution paths in the pool first based on specific criteria, such as branch probability, relevance to security, and avoiding unnecessary runtime complexity. For example, a heuristic approach could prioritize functions that call security critical functions that use arguments tainted by user input somewhere in execution history. This would prove a very Agile alternative and prove extremely useful in a continuous integration pipeline, where quick and efficient feedback is essential. Using the right heuristics, an engineer can quickly and efficiently discover security-critical bugs in software before it reaches production in an automated fashion.

This final alternative approach aims to integrate another technique entirely. Taint analysis is a technique used to track how input data changes and propagates throughout program execution using tagging. This could prove very powerful when combined with UC-SE. This could improve efficiency and precision by focusing analysis on paths affected by user-controlled input. This is particularly valuable in software testing and can easily prioritize the detection of many vulnerability classes, as vulnerabilities a tester would be interested in would result from user input. By filtering execution paths based on user-controlled taints, we can reduce redundant exploration and be more selective of symbolic constraints. This was directly tested in the reference J. Li, et al where a system called “ConcolicFuzz” was built. This software combines dynamic taint analysis and control-flow targeted input filtering, reducing the number of symbolic variables and improving code coverage. In this tool, they identified input bytes that influence control flow and filtered irrelevant symbols from their analysis. This, as their metrics will support, resulted in symbolic inputs that are likely to alter program paths, increasing coverage. Additionally, taint analysis helped avoid path explosion by not wasting effort on irrelevant paths. Having

a tool like this implemented into the testing phase could periodically triage code-related issues and frequently deliver high-impact wins to a team.

Evaluation Criteria

Let's take a moment to consider the evaluation criteria that would reflect a good combination of alternatives in addressing the issues with UC-SE testing. The most obvious and fundamental of which is scalability. A good solution should address this inherent issue by enabling a tool to safely handle large codebases with minimal path explosion. Scalability is a major challenge in developing any software but is especially prominent when symbolic execution gets thrown into the mix. Path explosion specifically poses a space-complexity issue as well as a large timeframe for processing. These issues do still exist in under-constrained initializations due to conditions wrapped by loops and recursive calls. In a software development context, this impacts debugging, testing, and security verification as developers need tools that can practically analyze large targets without excessive overhead. A scalable solution would optimize and carefully consider constraints, minimize redundant exploration, and integrate easily with modern development pipelines.

Precision is another critical criterion that should not be considered as a trade-off in developing an effective SE fuzzer. An engineer's use of UC-SE should not result in over-approximation and false positives. A good concolic analysis suite would limit the potential path pool by reducing infeasible paths and considering likely execution context. The potential pool along with all other considerations related to the actual analysis, should be considered before analysis actually occurs. A mitigation would aim to eliminate paths from being explored and prevent the caching of false positives. This issue can mislead developers by flagging non-existent issues and wasting time, or conversely, by missing critical vulnerabilities because of over-approximation. Poor precision results in poor software verification. A solution should refine analysis to reduce false alarms while ensuring that real defects are discovered and recorded accurately.

Agile compatibility should be considered as a significant criterion, and a good solution should integrate easily into existing Agile CI/CD pipelines. A solution should provide continuous feedback, wins, and be easy to use. It should integrate well into development frameworks that aim to provide incremental feedback and provide a basis for continuously improving the product being tested/maintained over time.

The last of these criteria involves context awareness. A solution should provide realistic execution modeling and accurate state simulation. The under-constrained nature of under-constrained analysis results in a lack of full program context during

analysis. Execution states leading up to the constrained site should be reasonable and related to realistic use-cases of the target. In other words, the analysis should reflect real world use and any pre-configurations at the constrained site such as memory values, should be considerate of this nature. Software behavior and analysis is highly dependent on the execution context of the target. A major limitation of under-constrained analysis, if not the most significant, is its lack of historical execution data, which can in turn lead to incorrect assumptions about behavior. In a development workflow, this affects debugging, regressive testing, and security analysis, as a developer would need to rely on that the tool in reflecting real-world execution. To make symbolic execution more applicable and capable in a software suite, solutions must use real-world constraints.

Evaluation Table

Alternative	Scalability	Precision	Agility	Context-Awareness
A. Precomputation	High Reduces paths.	Medium Possibly introduces over-approximations.	High Can be automated and integrated into workflows.	Medium Uses common states, but semi-contextless.
B. Selective Concretization	Medium Reduces symbolic complexity but also needs selection algorithm.	High Reduces infeasible paths significantly.	Medium Manual cases could exist.	Low Limits code coverage.
C. Priority Heuristics	High Reduces paths significantly.	Medium Possibly misses critical execution states.	High Works well in CI/CD pipelines.	Medium Does not consider history but prioritizes likely paths.
D. Taint Analysis	Medium	High	Low	High

	Slightly reduces paths but may still explore infeasible paths.	Focus on security-relevant paths.	Additional layer of complexity.	Uses realistic input-driven states to guide execution.

Justification of Evaluations

Precomputed initial states perform well in Agile environments since they allow for fast and repeatable testing across code blobs. In R. Rutledge and A. Orso, the ODIT tool uses UC-SE to initialize symbols utilizing internal program functions, enabling faster targeted analysis without a large context. This supports the real-world application of pre-computed symbolic contexts that effectively work in regression testing and can drive forward Agile pipelines.

Purposeful concretization is an excellent solution for precision, especially when targeting programs with inputs that have limited probabilities in their concrete values, but it sacrifices realism and can limit path coverage due to the mentioned reduced symbolism.

Heuristic-based path prioritization is perhaps the most scalable solution and aligns with Agile's need for fast and incremental feedback loops. In Yeh, C. et al, Monte Carlo Tree Search (MCTS), a tree-searching algorithm, is applied to prioritize symbolic execution paths and its effects on defeating path explosion and improving efficiency are explored. The solution provably reduces overhead and aligns well with the Agile principles. It is, however, important to note that the effectiveness of this solution depends greatly on the quality of heuristics chosen.

Taint analysis integration offers the most targeted and bug-centric solution by focusing on user-controlled data flow, which can be valuable for finding security-critical issues first. While it is highly effective for security-centric testing, it introduces another layer of complexity which may not be ideal in a fast-paced Agile environment.

Recommendation

An Agile solution should mainly focus on balancing scalability, practicality, and ability to integrate while maintaining a reasonable accuracy in accordance with the test results. With this in mind, I would recommend a hybrid solution combining both precomputed execution states and heuristic-based path selection. This balances

scalability and Agile integration. Pre-computation provides a reliable method of starting UC-SE at logical starting points such as critical functions, which would eliminate a great deal of overhead. This aligns with Agile principles by reducing analysis time. Furthermore, it would enable modular analysis units that can be of use elsewhere and implemented into CI/CD pipelines. Heuristics ensure that the engine explores high-value paths first and improves UC-SE without interacting with the complexity of SE itself. This would be the precision aspect of the solution. An example heuristic could include a heuristic that targets functions with high-user input, dynamic typing, or high-risk call sites in a program. This enables fast feedback to developers, as well as allowing for the continuous tuning of the solution to adapt and meet new software delivery cycles. While I recommend these two alternatives specifically, I do believe a security-centric team would benefit from additionally integrating taint analysis. While taint analysis introduces complexity and overhead, security-focused teams working on security-critical products could use this solution to more effectively focus on user-triggerable bugs. This solution should also be incorporated in situations where input validation or potential exploitability of a product is a central concern. By adapting this balanced recommendation into approaches, software teams can integrate under-constrained symbolic execution into modern development workflows while maintain precise, scalable, and actionable insights in a manner that does not compromise the speed, complexity or overall delivery of a product or continued maintenance thereof.

Bibliography

J. Li, X. Xu, L. Liao and L. Li, “Concolic Execute Fuzzing Based on Control-Flow Analysis,” 2015 11th International Conference on Computational Intelligence and Security (CIS), Shenzhen, China, 2015, 385-389.

Yeh, C., Lu, H., Yeh, J., & Huang, S. (2017). “Path Exploration Based on Monte Carlo Tree Search for Symbolic Execution,” *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, 2017, 33-37.

R. Rutledge and A. Orso, “Automating Differential Testing with Overapproximate Symbolic Execution,” 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), Valencia, Spain, 2022, 256-266.