National Satellite of Excellence in Trustworthy Software Systems
May 2021
URL: https://www.comp.nus.edu.sg/~nsoe-tss/

This document summarizes the core competencies of the National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) at Singapore. It consolidates the key achievements to date, including publications and tool releases. Overall queries about NSoE-TSS can be directed to *Dr. Gregory J Duck, Assistant Director, gregory@comp.nus.edu.sg*

The core competencies of the NSoE-TSS are:

1. Fuzzing and Symbolic Execution
2. Program Repair, Binary hardening/analysis
3. Artificial Intelligence testing
4. Formal Verification

Fuzzing and Symbolic Execution

Fuzzing is a proven and effective method for software testing, and is responsible for the detection of many security vulnerabilities and other bugs in modern software systems. At its core, fuzz testing can be seen as a biased random search over the domain of program inputs, with the goal of uncovering inputs that cause the program to crash or hang. Typically, fuzz testing generates thousands of inputs per second, which can more effectively stress the target program compared to traditional developer-provided test suites. Modern fuzzers also use feedback from the target, such as path coverage or symbolic information, in order to guide the search and make it more effective at discovering new bugs.

Fuzz testing is one of the core competencies of the NSoE-TSS project. However, most of the existing work on fuzz testing is applicable to traditional software domains, such as user-mode Linux programs with well defined inputs and outputs. The success of fuzz testing under this domain has inspired new applications. This report highlights several new domains for fuzz testing and related technologies, including Deep Neural Networks (DNNs), compilers.

Fuzzing: Reflections article. [NSoE-TSS core]
In this article, we have worked with our co-authors and collaborators worldwide to summarize the field of fuzzing and its challenges. The article was triggered by discussions in a Shonan Meeting in 2019 September. It has also been highlighted in the Shonan Meetings repository https://shonan.nii.ac.jp/publications/others/

Fuzzing: Challenges and Reflections, Marcel Boehme, Cristian Cadar, Abhik Roychoudhury, IEEE Software, 38(3), pages 79-86, 2021.

Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks [NSoE-TSS Core]

DNNs are notoriously brittle with respect to small perturbations in their input data. This problem is analogous to the overfitting problem in test-based program synthesis and repair. This problem is a consequence of the incomplete specification, i.e., the limited tests or training examples that the program synthesis or repair algorithm has to learn from. Recently, test generation techniques have been successfully employed to augment existing specifications of intended program behavior, to improve the generalizability of program synthesis and repair. Inspired by these approaches, we developed the Sensei and Sensei-SA tools based on techniques that repurpose software testing methods, specifically mutation-based fuzzing, to augment the training data of DNNs, with the objective of enhancing their robustness. Our technique casts the DNN data augmentation problem as an optimization problem, and uses genetic search to generate the most suitable variant of an input data to use for training the DNN, while simultaneously identifying opportunities to accelerate training by skipping augmentation in many instances. We evaluate our tools on 15 DNN models spanning 5 popular image data-sets. Our evaluation shows that Sensei can improve the robust accuracy of the DNN, compared to the state of the art, on each of the 15 models, by upto 11.9% and 5.5% on average. Further, Sensei-SA can reduce the average DNN training time by 25%, while still improving robust accuracy. This project was published in ICSE2020 and the tool is available here:

- https://sensei-2020.github.io/
- Xiang Gao, Ripon K. Saha, Mukul R. Prasad, Abhik Roychoudhury, *Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks*, International Conference on Software Engineering, 2020
- TRL4
- Target user: academia & industry

TimeMachine: an automated testing tool for Android apps [NSoE-TSS Core]

One of the key challenges in software testing for mobile applications. Specifically, Android testing tools generate sequences of input events to exercise the state space of the app-under-test. Existing search-based techniques systematically evolve a population of event sequences so as to reach certain objectives such as maximal code coverage. The hope is that the mutation of fit event sequences leads to the generation of even fitter event sequences. However, such random mutations may truncate the path of the generated sequence through the app's state space at the point of the first mutated event. States that contributed considerably to the original sequence's fitness may not be reached by the sequence's offspring. To address this challenge, we develop the notion of time-travel testing. The basic idea is to evolve a population of states which can be captured upon discovery and resumed when needed. The hope is that generating events on a fit program state leads to the transition to even fitter states. For instance, we can quickly deprioritize testing the main screen state which is visited by most event sequences, and instead focus our

limited resources on testing more interesting states that are otherwise difficult to reach. We have implemented our approach in the form of the TimeMachine testing tool for Android applications. Our experimental results show that TimeMachine outperforms the state-of-the-art search-based Android testing tools Sapienz and Stoat, both in terms of coverage achieved and crashes found. The project was published in ICSE 2020 and the tool is available here:

- https://github.com/DroidTest/TimeMachine
- Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury, *Time-travel Testing of Android Apps*, International Conference on Software Engineering, 2020
- TRL4-5
- Target user: academia and industry

OASIS: A Novel Hybrid Kernel Symbolic Execution Framework For Malware Analysis [NSOE-TSS2019-04, Prof. Ding Xuhua]

OASIS is a novel software system infrastructure for secure and transparent software/malware analysis across both user and kernel spaces. A software analyzer can develop her analysis applications to run on top of OASIS which provides the interfaces from them to introspect, modify and control a running target software (including the operating system kernel) in an on-demand fashion, without instrumenting it. Exemplary tools we showcase with OASIS include full-space tracer of a shell commands such as `ls`, and data flow analysis of a kernel device driver.

- https://github.com/OnsiteAnalysis/OASIS
- Jiaqi Hong, Xuhua Ding, *A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces*, Security and Privacy, 2021
- TRL4
- Target users: academia, industry

The NSOE-TSS2019-04 team are working on two follow-up tools that use OASIS, with plans to release before the end of the project:

- *KLEEN (KLEE-Native)*: KLEEN is a novel hybrid symbolic execution engine derived from the fusion of KLEE and OASIS.
- *KLEEK (KLEE-Kernel)*: KLEEK is expected to be the hybrid symbolic execution engine for kernel analysis.

SpecTest: Specification-based Compiler Fuzzing [NSOE-TSS2019-03, Prof. Sun Jun]

SpecTest is a novel compiler testing technique that targets less-used language features. SpecTest is based on three components: an executable language specification, a fuzzer for generating test inputs, and a mutator which generates new programs by injecting rare language features. SpecTest is an example of applying fuzz testing to new domains in order to uncover novel bugs, in this case compilers and programming language implementations. Comparing the abstract execution of the specification to the concrete execution of a compiled program enables

our method to find deep semantic errors as well as inconsistencies and issues in the specification. We already applied our tool in real-world use-cases, i.e. by testing the Solidity and Java compiler. We were able to identify bugs and issues in both compilers, which showed the applicability of our tool/approach.

- Richard Schumi, Jun Sun, *SpecTest: Specification-Based Compiler Testing*, International Conference on Fundamental Approaches to Software Engineering, 2021.
- Anticipated release: July 2021.
- Anticipated TRL: 4
- Anticipated target: academia

Program Repair and Binary Hardening

Computing systems, specifically software systems, are prone to vulnerabilities which can be exploited. One of the key difficulties in building trustworthy software systems - is the lack of specifications, or intended behavior, or a description of how the software system is supposed to behave. In our work, we have developed semantic analysis techniques to extract or discover specifications from an erroneous or vulnerable program. Such a specification discovery process helps in automatically generating repairs, thereby moving closer to the goal of self-healing software systems. As more and more of our daily functionalities become software controlled, and with the impending arrival of technology like personalized drones, the need for self-healing software has never been greater. Program repair technology often uses program synthesis in the background to generate a patch automatically.

Moreover, software components are often distributed without source code in order to protect intellectual property rights and to discourage reverse engineering. Such *Commercial Off-the-Shelf* (COTS) software is ubiquitous, ranging from everything including closed-source programs, libraries, as well as embedded firmware in IoT and networking hardware. COTS is part of most modern software stacks, including those used by critical infrastructure, such as 5G mobile networks. COTS software may be untrusted, contain security vulnerabilities, or may be the victim of malicious interference such as supply chain attacks. The lack of source code makes binary rewriting/repair very challenging. The core NSoE-TSS team and grant recipients are developing technologies to help secure untrusted source and binary components through a combination of analysis, hardening and repair.

ExtractFix: Program Vulnerability Repair via Crash Constraint Extraction [NSoE-TSS Core]

Automated program repair techniques are typically driven by a correctness criterion based on test-suites, which is prone to generating overfitting patches (i.e., pass the tests but fail in production). To help address this issue, the NSoE-TSS Core team has developed ExtractFix---a repair tool which fixes program vulnerabilities without the need for a voluminous test-suite. Given a vulnerability as evidenced by an exploit, ExtractFix extracts a constraint representing the vulnerability with the help of sanitizers. The extracted constraint serves as a proof obligation which our synthesized patch should satisfy, which is met by propagating the extracted constraint to locations which are deemed to be "suitable" fix locations. ExtractFix is built on top of the KLEE symbolic execution engine (whitebox fuzzing).

- Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, Abhik Roychoudhury, *Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction*, Transactions on Software Engineering and Methodology (TOSEM), 2021
- TRL3
- Target user: academia

Fix2Fit: Crash-avoiding Program Repair [NSoE-TSS Core]

Fuzz testing can also be applied to *Automated Program Repair* (APR). Specifically, we aim to address the "overfitting" problem, which is one of the core challenges for APR. The overfitting problem occurs when the generated repair/patch satisfies the given test suite, but does not implement the correct behaviour for inputs outside of the test suite. The repaired program may not satisfy even the most basic notion of correctness, namely crash-freedom. To address this issue, our Fix2Fit tool integrates fuzz testing into a program repair framework in order to detect and discard crashing patches. Our approach fuses test and patch generation into a single process, in which patches are generated with the objective of passing existing tests, and new tests are generated with the objective of filtering out over-fitted patches by distinguishing candidate patches in terms of behavior. We use crash-freedom as the oracle to discard patch candidates which crash on the new tests. At its core, our approach defines a grey-box fuzzing strategy that gives higher priority to new tests that separate patches behaving equivalently on existing tests. This test generation strategy identifies semantic differences between patch candidates, and reduces over-fitting in program repair. We evaluated our approach on real-world vulnerabilities and open-source subjects from the Google OSS-Fuzz infrastructure. We found that our tool Fix2Fit (implementing patch space directed test generation), produces crash-avoiding patches. While we do not give formal guarantees about crash-freedom, cross-validation with fuzzing tools and their sanitizers provides greater confidence about the crash-freedom of our suggested patches. The Fix2Fit project highlights how technologies such as fuzzing can be combined and applied to other domains such as program repair, which are both relevant to the overall objectives of the project. Furthermore, program repair is relevant to the year 2 deliverables of which we have already made progress.

- https://github.com/gaoxiang9430/Fix2Fit
- Xiang Gao, Sergey Mechtaev, Abhik Roychoudhury, *Crash-avoiding Program Repair*, International Symposium on Software Testing and Analysis, 2019
- TRL4
- Target user: academia

Binary Rewriting without Control-Flow Recovery with E9Patch [NSoE-TSS Core]

The backbone of any binary hardening or repair project is the ability to accurately rewrite binary software in order to add/remove functionality. However, binary rewriting is notoriously difficult, with most existing solutions failing to scale to real-world software. To address this issue, the core NSoE-TSS team has developed the *E9Patch binary rewriting tool*.

E9Patch is the first truly scalable binary rewriting tool for x86_64 ELF binaries. E9Patch is based on a combination of novel instruction patching techniques, such as punning, padding and eviction, that operates at the machine code level. These techniques allow E9Patch to insert jumps to trampoline code (implementing some intended binary repair or hardening code) without the need to move existing instructions, thereby preserving the original control flow of the input binary. E9Patch does not need to recover control flow information through static analysis, heuristics or assumptions, thereby eliminating one of the major sources of incompatibility that plague existing state-of-the-art static binary rewriting systems. As such, E9Patch can readily scale to very large binaries (>100MB) including web browsers such as Google Chrome and Firefox.

- https://github.com/GJDuck/e9patch
- Gregory J. Duck, Xiang Gao, Abhik Roychoudhury, *Binary Rewriting without Control Flow Recovery*, Programming Language Design and Implementation (PLDI), 2020
- Target users: academia and industry
- TRL7+

Several projects based on the E9Patch technology are under development, including:

- e9afl: binary fuzzing [https://github.com/GJDuck/e9afl]
- e9syscall: system call interception for environment modeling [https://github.com/GJDuck/e9syscall]
- redzone+lowfatptr: hardening binaries against memory errors using complimentary redzone and low-fat-pointer protection.
- binary repair project: *Automatic Binary Repair* (ABR) based on *Automatic Program Repair* (APR) technologies

Improved function signature extraction for optimized binaries [NSoE-TSS2019-02, Prof. Debin Gao]

Control-Flow Integrity (CFI) is a method for program hardening that defends against control-flow hijacking attacks which are commonly used to exploit vulnerable programs. CFI works by enforcing the program's Control-Flow Graph (CFG) at runtime, meaning that any attempt to divert control-flow to attacker-controlled code will be detected. Hardening binaries with CFI is particularly challenging, given that the CFG can be difficult to extract using analysis, especially for stripped binaries that have been compiled with optimization (-O2/-O3).

To help address the problem of accuracy for binary CFI enforcement, the *NSoE-TSS2019-02* team has developed *FSROPT*---an improved function signature extraction tool for optimized binaries. The tool consists of several components, such as:

1. *Ground Truth Collection*. It is developed as an LLVM pass, which will collect the ground truth of the function signatures including the number and types of arguments for each function and indirect caller.

2. *Recovered Function Signature Comparison*. A comparison between the SMU Classification, an existing tool (TypeArmor), and CFG information from the collected ground truth.
3. *Improved Function Signature Recovery Policy Enforcement*. Implements the improved policy to recover function signatures for each callee and indirect caller based on Dyninst.

Further information:

- https://github.com/ylyanlin/FSROPT
- Yan Lin, Debin Gao, *When Function Signature Recovery Meets Compiler Optimization*, Security and Privacy, 2021
- TRL5
- Target users: academia.

The NSoE-TSS2019-02 is currently developing two additional projects that are anticipated to be released by the 22nd March 2022 with TRL5:

- *End-to-end machine learning with domain knowledge for function signature extraction*: This component makes use of word embedding and three-layers Recurrent Neural Network (RNN) to infer the number and type of arguments for each function and caller.
- *Binary rewriting of Linux executables with CFI enforcement*: This component inserts the CFI policy data structure and corresponding CFI enforcement code into the original executable.

Artificial Intelligence Testing

Artificial Intelligence (AI) has achieved tremendous success in many applications. However, similar to traditional software, AI models can also contain defects, such as adversarial attacks, backdoor and noisy labels. Quality assurance of AI systems is needed. Artificial Intelligence is one of the core competencies of the NSoE-TSS project towards building trustworthy AI systems. This report highlights the techniques for evaluating and enhancing the trustworthiness of AI systems including runtime monitor, repair, noisy label handling and adversarial example detection.

SelfChecker: Self-Checking Deep Neural Networks in Deployment [NSOE-TSS2019-05, Prof. Dong Jin Song]

SelfChecker is a self-checking system that monitors DNN outputs and triggers an alarm if the internal layer features of the model are inconsistent with the final prediction. SelfChecker also provides advice in the form of an alternative prediction. SelfChecker uses kernel density estimation (KDE) to extrapolate the probability density distributions of each layer's output by evaluating the DNN on the training data. Based on these distributions, the density probability of

each layer's outputs can be inferred when the DNN is given a test instance. SelfChecker measures how the layer features of the test instance are similar to the samples in the training set. If a majority of the layers indicated inferred classes that are different from the model prediction, then SelfChecker triggers an alarm. In addition, not all layers can contribute positively to the final prediction. SelfChecker therefore uses a search-based optimization to select a set of optimal layers to generate a high quality alarm and advice.

- https://github.com/self-checker/SelfChecker
- Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, Jin Song Dong, *Self-Checking Deep Neural Networks in Deployment*, International Conference on Software Engineering, 2021
- TRL4

Improving Trustworthiness of Real-world AI systems through Adversarial Attack and Effective Defense [NSOE-TSS2019-01, Prof. Bo An]

The NSOE-TSS2019-01 project aims to provide a robustness evaluation service for deep learning models. Users only need to provide the SDK or API interface of the model, and select the dataset which is built into the platform or uploaded by themselves. Then the platform will generate adversarial attacks based on multiple algorithms. According to the changes of model accuracy under the attacks with different algorithms, iterations, and intensities, the tool will output a robustness score and detailed evaluation report.

Provably Consistent Partial-Label Learning [NSOE-TSS2019-01, Prof. Bo An]
We propose the first generation model of candidate label sets, and develop two novel PLL methods that are guaranteed to be provably consistent, i.e., one is risk-consistent and the other is classifier-consistent. Partial-label learning (PLL) is a multi-class classification problem, where each training example is associated with a set of candidate labels.
Even though many practical PLL methods have been proposed in the last two decades, there lacks a theoretical understanding of the consistency of those methods—none of the PLL methods hitherto possesses a generation process of candidate label sets, and then it is still unclear why such a method works on a specific dataset and when it may fail given a different dataset. Our methods are advantageous, since they are compatible with any deep network or stochastic optimizer. Furthermore, thanks to the generation model, we would be able to answer the two questions above by testing if the generation model matches given candidate label sets.

- Lei Feng, Jiaqi Lv, Bo Han, Miao Xu, Gang Niu, Xin Geng, Bo An, Masashi Sugiyama. Provably consistent partial-label learning. Proceedings of the Thirty-fourth Annual Conference on Neural Information Processing Systems (NeurIPS'20)
- Anticipated release date: 30/4/2022
- Anticipated TRL: TRL4
- Target users: industry and academic

Computing Ex Ante Coordinated Team-Maxmin Equilibria in Zero-Sum Multiplayer Extensive-Form Game [NSOE-TSS2019-01, Prof. Bo An]

We focus on computing the multiplayer Team-Maxmin Equilibrium with Coordination device (TMECor) in zero-sum extensive-form games. TMECor models scenarios when a team of players coordinates ex ante against an adversary. Such situations can be found in card games (e.g., in Bridge and Poker), when a team works together to beat a target player but communication is prohibited; and also in real world, e.g., in forest-protection operations, when coordinated groups have limited contact during interdicting illegal loggers. The existing algorithms struggle to find a TMECor efficiently because of their high computational costs. To compute a TMECor in larger games, we make the following key contributions: (1) we propose a hybrid-form strategy representation for the team, which preserves the set of equilibria; (2) we introduce a column-generation algorithm with a guaranteed finite-time convergence in the infinite strategy space based on a novel best-response oracle; (3) we develop an associated-representation technique for the exact representation of the multilinear terms in the best-response oracle; and (4) we experimentally show that our algorithm is several orders of magnitude faster than prior state-of-the-art algorithms in large games.

- Youzhi Zhang, Bo An, Jakub Cerny. Computing ex ante coordinated team-maxmin equilibria in zero-sum multiplayer extensive-form games. Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)
- Anticipated release date: 30/4/2022
- Anticipated TRL: TRL4
- Target users: industry and academic

Formal Verification

Formal Verification is a well known technique to prove the correctness of a system, and during the last decades has been successfully used to demonstrate that a system fulfills a set of safety and liveness properties. While other techniques only explore part of the domain under analysis, making it inefficient specially for concurrent systems, formal verification usually aims to cover the whole state space in order to guarantee the correctness of the system. One possible approach is model checking, which automatically explores the state space of a model representing the system to check if it is correct. However this approach often does not scale well for large systems. In that case, it is possible to use deductive verification, which uses theorem provers to generate proof obligations that can be proven automatically often with the help of SMT solvers, or in the case of complex cases guided by experts. While deductive verification is more expensive, it is especially suited for the verification of high assurance systems, where their correctness is critical.

Formal Verification is a core competency of the NSoE-TSS project. In particular, formal verification in the NSoE-TSS focuses on three different aspects: first, the automation of deductive verification of concurrent systems, specially suited for micro-kernel verification; second,

procedures to guarantee the memory safety of programs; and third verification of distributed systems.

Automated Verification Condition Generation for Rely-guarantee properties on CSimpl models [NSoE-TSS Core]

We have developed a sound comprehensive set of rules for the Verification Condition Generator (VCG) for CSimpl. Giving a rely-guarantee specification for a program C, composed of a pre-condition, post-condition, and rely and guarantee relations, the VCG automatically calculates the weakest precondition resulting from applying backward reasoning rules on the program C from the postcondition. The set of developed rules provide a set of Rely-Guarantee rules to be used in single-step structured proofs in Isabelle/Isar. These rules allow assertions of Rely-Guarantee logic to be manipulated in calculational proofs, with the inclusion expressed in terms of sets or predicates, as well as reversed order. These rules are fundamental for the decision procedure that generates from the current postcondition and the CSimpl instruction, the weakest precondition. Once the weakest precondition has been calculated, we can infer that the specification is correct if the precondition of the specification implies the calculated weakest precondition.

This set of rules includes first different representations of the consequence rules, including the Kleymann representation. Then we provide the rules for each CSimpl construct: Skip, Basic, Spec, Cond, While, Throw, Seq, Catch, Guard, Await, Call, and Dyncom. From these, we also provide rules for composed constructs such as raise, condCatch, bind, bseq, block, dynCall, fcall, and switch.

Second, we have developed a sound comprehensive set of rules for the VCG for CSim, the simulation framework for the preservation of Rely-Guarantee properties in CSimpl. Similarly to the VCG for CSimpl, the VCG for CSim uses backward calculation to obtain the weakest precondition relation given a precondition and post-condition relations relating a specification and implementation given a relation α between them. We develop a set of rules to express in weakest precondition form the inference system of CSim. Then we implement a decision procedure in the functional language ML, the implementation language in the Isabelle HOL theorem prover.

- Paper accepted **CSim2 : Compositional Top-down Verification of Concurrent Systems Using Rely-Guarantee**. David Sanan, Yongwang Zhao, Shang-Wei Lin, Yang Liu. ACM Transactions on Programming Languages and Systems. January 2021.
- Anticipated release date: 31/12/2021
- Anticipated TRL: TRL4
- Target users: academic

Core Language for systems programming [NSoE-TSS Core]

We use the semantics of OSL to model the C language to guarantee that programs written in C are memory safe if there is a possible translation between C and OSL considering translational rules that preserve the ownership system. If the input program is not memory safe then K-OSL

stops in an intermediate configuration where the program is not fully processed and leading to potential unsafe memory risks.
One of the objectives for OSL is the application of the ownership system to other languages to guarantee memory safety on programs written on those languages. The main challenge is that whilst OSL labels each variable and each reference as mutable or not, C misses such information. There are two approaches to solve this: (1) add annotations to the statements in the C program including mutability information, (2) design an automatic decision procedure to infer mutability information of variables and references. To use the OSL in the verification of other programming languages, it is necessary to define a set of translation rules to build the correspondence between the programming language and OSL.

To have a complete set of rules, we design a complete set of rules covering C static and dynamic variables, and referencing and dereferencing expressions leading to mutable shareability. The rules also cover Const variable declarations in C, introducing the concept of immutability. From this, we add the necessary rules to extract information about mutability and immutability for function arguments. Also, the transformation procedure adds rules for volatile and restrict modifiers. To improve the performance of the application, we add heuristics from the semantics of memory related functions like memcpy, and rules for static and dynamic arrays.

- Paper under submission
- Anticipated release date: 31/12/2021
- Anticipated TRL: TRL4
- Target users: academic and industry.

Trustworthy Distributed Software with Safety and Liveness Guarantees [NSOE-TSS2019-06, Prof. Chin Wei-Ngan]

The high-level objective of this project is to bring the scientific principles of rigorous mathematical specification and verification at the core of the distributed software design and implementation with respect to the following properties: functional correctness, safety and liveness. More specifically, we aim to (1) design a series of expressive, unambiguous and readable specification logics/languages in which the software systems can be described and analysed; (2) design modular and automatic verification systems guided by formal specifications to check for the safety and the functional correctness of the implementation; and (3) implement a prototype to show the feasibility of our project and use the outcomes of (1) and (2) for real-world applications

We have thus far tackled objective (1) and (2) and part of (3). We first worked on designing a specification logic which can be used to unambiguously describe distributed applications. The logic is expressive enough to describe how the components of the application interact with each other both functionally as well as in terms of their order relation. Most of the previous approaches in the space of distributed applications specifications have either used ambiguous specifications which come with little support for formal guarantees (e.g. RTF, XML), or have designed highly complex languages with strong correctness guarantees but which require considerable expertise to manipulate them [1,2,3], or approaches which are easier to use, but are too abstract to offer

strong guarantees [4,5,6,7,8]. We aimed for a compromise between the latter two, and have designed a logic which is inspired by multiparty session types [8], thus offering a friendly interaction with the user, while maintaining mathematical rigor needed to support strong safety and correctness guarantees. We have achieved this by combining the syntax of session types with the expressivity of separation logic [9]. The specification language is used to guide a Hoare-style modular verification system.
Furthermore, we have also investigated the integration of temporal properties into program specifications. Existing approaches to temporal verification have either sacrificed compositionality in favour of achieving automation or vice-versa. To exploit the best of both worlds, we present a new solution to ensure temporal properties via a Hoare-style verifier and a term rewriting system (TRS) on integrated dependent effects.

Another avenue we have started to investigate is that of security protocols which are an important class of distributed applications. In this direction we have examined tools like Tamarin and Proverif which represent the state of the art in protocol analysis for means to analyse for the authenticity and confidentiality of protocols. Our survey concluded that while these tools have good results at the level of protocol analysis, there is no guarantee offered with respect to their implementation. We would like to cover some of this uncharted territory that focuses on ensuring correct distributed applications. In a country such as Singapore which is moving towards a smart city, ensuring security properties on top of safety is of utmost importance. Additionally, we have investigated common explicit synchronisation mechanisms used in Go and Java.

Released tools

- http://loris-5.d2.comp.nus.edu.sg/Mercurius
- https://github.com/songyahui/EFFECT
- https://github.com/songyahui/SyncedEffects
- https://hub.docker.com/u/hippodrome
- https://nem-repair-tool.github.io

Publications

- Nguyen, Thanh-Toan, et al. "Automated Repair of Heap-Manipulating Programs Using Deductive Synthesis." International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, Cham, 2021

CertiChain: A Framework for Mechanically Verifying Blockchain Consensus Protocols [NSOE-TSS2019-07, Prof. Ilya Sergey]

Sharding is a popular way to achieve scalability in blockchain protocols, increasing their throughput by partitioning the set of transaction validators into a number of smaller committees, splitting the workload. Existing approaches for blockchain sharding, however, do not scale well

when concurrent transactions alter the same replicated state component - a common scenario in Ethereum-style smart contracts.

We propose a novel approach for efficiently sharding such transactions. It is based on a folklore idea: state-manipulating atomic operations that commute can be processed in parallel, with their cumulative result defined deterministically, while executing non-commuting operations requires one to own the state they alter. We present CoSplit - a static program analysis tool that soundly infers ownership and commutativity summaries for smart contracts and translates those summaries to sharding signatures that are used by the blockchain protocol to maximise parallelism. Our evaluation shows that using CoSplit introduces negligible overhead to the transaction validation cost, while the inferred signatures allow the system to achieve a significant increase in transaction processing throughput for real-world smart contracts.

- Certifying Certainty and Uncertainty in Membership Query Structures, Kiran Gopinathan and Ilya Sergey, CAV 2020.
- TRL4
- Academia