Kick-off: Software Security Analysis

Chair for IT Security / I20
Prof. Dr. Claudia Eckert
Technical University of Munich

Dr. Julian Schütte
`julian.schuette@aisec.fraunhofer.de`

5.2.2020

Outline

Requirements

The seminar will be organized as a scientific conference. You will present your research in written and in a presentation to your peers.

The paper you will be writing will (most likely) be a *Systematization of Knowledge (SoK)* or *introductory* paper.

SoK papers do not propose a novel approach. They take a broader view on a topic, explain the core concepts and put the most relevant works in context.
Introductory papers explain the core concepts of a field, the problems they are applied to and ongoing research directions.

Requirements

- Research & Paper Writing
  - Write a scientific paper of (exactly) 10 pages (including references and appendices)
  - English is recommended
  - We will provide a LaTeX template
- Review Phase
  - Every participant creates 2-3 reviews of her/his peers
  - Review template will be provided
  - 1 page
- Rebuttal & "Camera Ready" Phase
  - Integrate the reviewer's remarks, improve your paper as far as possible
  - Submit the "camera ready" version (final polished version)
  - Write a *rebuttal*, i.e. a response summarizing how you addressed the reviewers' remarks
- Presentation
  - 30 minutes presentation (English recommended, but German is also okay)
  - 15 minutes discussion

Time Table

05.02.2020	[Today] Topic Presentations. *Register for this seminar until 12.02.2020.*
20.02.2020	Start of topic assignments (once matching is finished in TUMonline)
17.04.2020	Submit your first version (outline finished, 80% of content)
22.04.2020	**Meeting**: Joint intermediate review and discussion
09.06.2020	Submit your paper
10.06.2020	Receive papers for review
16.06.2020	Submit your reviews
18.06.2020	Receive your reviews
25.06.2020	Submit your rebuttal + "camera-ready" version + presentation
02.+03.07.2020	**Meeting**: Presentations and discussion

Requirements

"First version"	Structure & main contents of the paper are fix. Introduction, conclusion, abstract might not be fully finished. Language does not have to be perfect, graphics might not be finished, some references might be missing. Focus on the "meat" of the paper!
"Rebuttal"	Your answer to the reviewer. Explain which suggestions you incorporated in the final version. If you do not agree with any suggestions, provide a short justification.
"Camera Ready"	The *perfect* and final version of your paper that you and your reviewers will be happy with. Correct formatting, correct citations, no typos.

Grading

The grading is composed of *mandatory* and *graded* parts:

Mandatory:

1. Timely submission of paper, reviews, final paper
2. Participation in discussions

Graded:

1. Paper (70%)
   - "First version"
   - "Final version"
   - "Rebuttal" & "Camera-ready" $\leftarrow$ focus of grading
2. Presentation (30%)

The next sessions will take place at Fraunhofer AISEC building.
Room will be announced.

Fraunhofer AISEC, TUM Campus, Lichtenbergstr. (opposite of Café Lichtenberg)

Topics (Overview) I

1. Program Query Languages
2. Attacking Software with Advanced Fuzzing
3. Graph-Based Software Analysis
4. Weighted Pushdown Systems as a Generic Analysis Framework
5. Finding Vulnerabilities with Symbolic and Concolic Execution
6. Abstract Interpretation as a Security Bug-Catching Technique
7. Dynamic Binary Instrumentation
8. Detecting Cryptographic API Misuse
9. Challenges of Interprocedural Analysis
10. Discover Security Vulnerabilities through Machine Learning supported Static Analysis
11. Vulnerability Discovery through Machine Learning supported Fuzzing
12. Discover Privacy Violations in Mobile Apps
13. Hybrid Analysis: Overcoming Limitations of Static and Dynamic Analysis
14. Automatic Proof Generation for Software Security Verification
15. Language-Aided Static Analysis

Program Query Languages I

- Programs can be regarded as a database of facts
- In this case, static analysis boils down to writing queries against the program in a *program query language*
- Possible contents of the paper: review existing query languages for security analysis of programs and discuss their differences, advantages, and deficits
    - Purpose? Generic vs. specific languages
    - On what representation of the program does the language operate?
    - Expressiveness? To what kinds of static analysis does the language correspond?

Program Query Languages II

References

- Martin et al. Finding application errors and security flaws using PQL. 2005
- Avgustinov et al. Variant analysis with QL. 2018
- CodeQL: `https://securitylab.github.com/research`
- RustQL: `https://github.com/rust-corpus/rustql`
- Johnson & Simha. CRAQL: A Composable Language for Querying Source Code. 2019
- Foo et al. SGL: A domain-specific language for large-scale analysis of open-source code. 2018
- Krüger et al. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. 2019

Attacking Software with Advanced Fuzzing I

- Software fuzzing is a technique of feeding random input data into a program and observe its execution for abnormal behavior, indicating a vulnerability
- Despite its simplicity, fuzzing is surprisingly effective and has discovered various high-profile vulnerabilities in the past
- The "secret sauce" of fuzzing is to efficiently cover "interesting" execution paths. There are various approaches on guiding the fuzzer to relevant code locations by observing the software under test
- Possible contents of this paper: review and categorize approaches on advanced fuzzing: coverage-based, statically guided, symbolically-assisted & machine learning-assisted fuzzing
- Note: limit the paper to fuzzing of *programs*

Attacking Software with Advanced Fuzzing II

References

- American Fuzzy Lop – a security-oriented fuzzer.
  `https://github.com/google/AFL`
- Bekrar et al. A taint based approach for smart fuzzing. 2012.
- Haller et al. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. 2013
- Cha et al. Program-adaptive mutational fuzzing. 2015
- Stephens et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution, 2017.
- Karamcheti et al. Adaptive grey-box fuzz-testing with thompson sampling. 2018.

Graph-Based Software Analysis I

- Graph-based representations are one of the main building blocks of program analysis
- Typically, different graphs are created for specific purposes (DDG, CFG, CG), but some researchers also regard the graph representation itself as a database that contains information about vulnerability patterns and can be queried.
- Possible contents of this paper:
  - Introduce basic graph representations used in program analysis
  - Introduce the generic framework of property graphs
  - Review literature & discuss their approaches of using graph queries for program analysis
  - Bring in your own view. Where do you see challenges/obstacles?

Graph-Based Software Analysis II

References

- Pewny et al. Cross-architecture bug search in binary executables. 2015.
- Yamaguchi et al. Modeling and discovering vulnerabilities with code property graphs. 2014.
- Yamaguchi et al. Automatic inference of search patterns for taint-style vulnerabilities. 2015.
- Schütte & Titze: liOS: Lifting iOS Apps for Fun and Profit. 2019

Weighted Pushdown Systems as a Generic Analysis Framework I

- Weighted Pushdown Systems (WPDS) allow to reason about properties of a program polynomial complexity
- Can be used to implement different types of analyses: context-sensitive data flow analysis, typestate analysis
- Are an alternative to the more common graph reachability-based approach for data flow analysis[1]
- Task: Understand WPDS and their application for program analysis. Consider different analysis types that can be "plugged" into the framework. Give an overview of how they can be used to detect security vulnerabilities.

Weighted Pushdown Systems as a Generic Analysis Framework II

References

- Reps et al. Weighted pushdown systems and their application to interprocedural dataflow analysis. 2005
- Lal et al. Extended Weighted Pushdown Systems. 2010
- Liang et al. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. 2013
- Song et al. Pushdown model checking for malware detection. International Journal on Software Tools for Technology Transfer. 2014
- Balakrishnan et al. Model checking x86 executables with CodeSurfer/x86 and WPDS++. 2005
- Späth et al. Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. 2019

[1] Reps et al. Precise interprocedural dataflow analysis via graph reachability. 1995

Finding Vulnerabilities with Symbolic and Concolic Execution I

- Symbolic Execution (SymEx) determines how inputs affect program execution
- Therefore, the analysis operates on symbolic values and replace concrete program executions with manipulations to these values
- In the last years, SymEx gained lots of attention

⇒ There are tons of (Open Source) tools available!

- In this paper, summarize the ideas of SymEx, which challenges arise from the most trivial implementation (e.g. path explosion, efficiency issues, analysis of complex program parts) and how these have been addressed in the last years

Finding Vulnerabilities with Symbolic and Concolic Execution II

References

- King. Symbolic Execution and Program Testing. 1976
- Cadar & Sen. Symbolic execution for software testing: Three decades later. 2013
- Baldoni et al. A survey of symbolic execution techniques. 2018
- Schwartz et al. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). 2010
- Cadar et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. 2008
- Sen et al. CUTE: A concolic unit testing engine for C. 2005
- Vitaly Chipounov. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. 2014
- Chau et al. SymCerts: Practical Symbolic Execution For Exposing Noncompliance in X.509 Certificate Validation Implementations. 2017
- `https://angr.io/`

Abstract Interpretation as a Security Bug-Catching Technique I

- Abstract Interpretation: static analysis technique, coming from compiler optimization
- Idea: Find an abstract model to approximate the program and prove that certain constraints hold.
- Applications in software security analysis e.g. integer overflows, buffer overflows
- Possible contents of the paper: explain the theoretical foundations of Abstract Interpretation, discuss its applications to security and put a focus on practical implementations.

Abstract Interpretation as a Security Bug-Catching Technique II

References

- Cousot and Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. 1977
- Allamigeon and Hymans. Static analysis by abstract interpretation: application to the detection of heap overflows. 2008
- Brat et al. IKOS: a Framework for Static Analysis based on Abstract Interpretation. 2014
- Gershuni et al. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. 2019
- Wang et al. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. 2019

Dynamic Binary Instrumentation I

- DBI allows to hook into running processes and inspect & modify execution
- This can happen in different ways: probe/hook injection, Dynamic Binary Translation: JIT compilation vs. ISA translation

- Possible content of this paper:
  - Explain the purpose and the design space of DBI
  - Discuss how DBI is used for security analysis. How does it relate to static analysis techniques?
  - Juxtapose it with related approaches such VMI
  - Review existing tools (Frida, Pin, Valgrind, DynamoRio) and discuss their design choices

Dynamic Binary Instrumentation II

References

- Frida `https://frida.re/`
- Backes et al. You can run but you can't read: Preventing disclosure exploits in executable code. 2014
- Nethercote. Valgrind: A framework for heavyweight dynamic binary instrumentation. 2007
- Zeng et al. PEMU: A pin highly compatible out-of-VM dynamic binary instrumentation framework. 2015
- D'Elia et al. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). 2019

Cryptographic API Misuse I

- Implementing your own cryptography is a *very* bad idea
- But turns out that even *using* cryptographic libraries is error-prone
- Automatically checking code for wrong usage of cryptographic libraries is an important research topic
- Possible content of the paper:
  - Introduce typical mistakes/vulnerabilities when using common cryptographic libraries
  - Review existing tools to detect these vulnerabilities and discuss the underlying analysis techniques.
    - What are their limitations?
    - What are their strengths?

Cryptographic API Misuse II

References

- CogniCrypt `https://www.eclipse.org/cognicrypt/`
- Rahaman et al. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. 2019
- Afrose et al. CryptoAPI-bench: A comprehensive benchmark on java cryptographic API misuses. 2019

Challenges of Interprocedural Analysis I

- Static analysis relies on a model of the code
- Many solutions exist to analyze intraprocedural programs
- Reality: Function calls are the rule, not the exception
- They affect the analysis results and require a proper model
- This exposes new challenges to analysis: Model function calls, explosion of states and branches
- Possible contents of this paper:
  - Summarize challenges arising from interprocedural analysis
  - Review how theoretical and practical frameworks address them

Challenges of Interprocedural Analysis II

References

- Reps et al. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95)
- Reps et al. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In International Static Analysis Symposium 2003
- Khedker and Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In Compiler Construction 2008
- Padhye et al. Interprocedural data flow analysis in Soot using value contexts. In Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis (SOAP '13)
- Späth et al.Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. Proceedings of the ACM on Programming Languages, (POPL '2019)

Discover Security Vulnerabilities through Machine Learning supported Static Analysis I

- Software projects are becoming larger in size and in number
- Large scale manual auditing becomes less feasable
- Idea: Mine open source code and known vulnerabilities, use machine learning techniques to train and classify pieces of code as potentially vulnerable.
- Is enabled by decades of vulnerability documentation and easy to access OpenSource code.
- Give an overview on Static Code Analysis supported by Machine Learning and the used technique to detect vulnerabilites. Does the appraoch need human interaction? What work has to be done to find a vulnerability.

Discover Security Vulnerabilities through Machine Learning supported Static Analysis II

References

- Yamaguchi et al. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. 2011
- Ghaffarian et al. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. 2017
- Zhou et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. 2019

Vulnerability Discovery through Machine Learning supported Fuzzing I

- Consists of repeatedly testing an application with modified inputs with the goal of finding security vulnerabilities.
- The input grammar has to automatically generated and adapted.
- Idea: Use Machine Learning to guide the process of input generation.
- Give an overview on Fuzzing and the use of different ML techniques to support the process, find and discuss more related work.

Vulnerability Discovery through Machine Learning supported Fuzzing II

References

- Godefroid et al. Learn&fuzz: Machine learning for input fuzzing. 2017
- Yan, et al. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. 2017
- Grieco et al. Toward Large-Scale Vulnerability Discovery using Machine Learning. 2016

Discover Privacy Violations in Mobile Apps

- Identify and list personally identifiable information that can be collected on Andoid and/or iOS
- Collect and evaluate methods for automated analysis of mobile apps to identify privacy concerns (e.g. Taint analysis)
- Describe counter-measures built into Android/iOS and/or provided by third party apps

- Initial literature
  - Mumtaz et al.: Critical review of static taint analysis of android applications for detecting information leakages
  - Enck et al.: TaintDroid. An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones
  - Egele et al.: PiOS. Detecting Privacy Leaks in iOS Applications
  - Wang et al.: DroidContext. Identifying Malicious Mobile Privacy Leak Using Context
  - Wu et al.: Efficient Fingerprinting-Based Android Device Identification With Zero Permission Identifiers

Fraunhofer AISEC

Hybrid Analysis: Overcoming Limitations of Static and Dynamic Analysis I

- Traditionally, program analysis is either static or dynamic
- However, both suffer from severe limitations
- A common application area of combining static and dynamic analysis is the automated generation of test cases for high code coverage
- Possible contents of this paper:
  - Summarize the limitations of classic static and dynamic analysis
  - Present an overview of hybrid program analysis techniques
  - Examine whether and how they solve the aforementioned limitations
  - What could be pitfalls of combinations of static and dynamic analysis?

Hybrid Analysis: Overcoming Limitations of Static and Dynamic Analysis II

References

- Barany and Signoles. Hybrid Information Flow Analysis for Real-World C Code. 2017
- Zhang. Palus: a hybrid automated test generation tool for java. 2011
- Zamfir and Candea. Execution synthesis: a technique for automated software debugging. 2010
- Du. Towards Building a Generic Vulnerability Detection Platform by Combining Scalable Attacking Surface Analysis and Directed Fuzzing. 2018
- Godefroid et al. Automated Whitebox Fuzz Testing. 2008

Discover Security Vulnerabilities through Machine Learning supported Static Analysis I

- Software projects are becoming larger in size and in number
- Large scale manual auditing becomes less feasable
- Idea: Mine open source code and known vulnerabilities, use machine learning techniques to train and classify pieces of code as potentially vulnerable.
- Is enabled by decades of vulnerability documentation and easy to access OpenSource code.
- Give an overview on Static Code Analysis supported by Machine Learning and the used technique to detect vulnerabilites. Does the appraoch need human interaction? What work has to be done to find a vulnerability.

Discover Security Vulnerabilities through Machine Learning supported Static Analysis II

References

- Yamaguchi et al.. 2011. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In Proceedings of the 5th USENIX conference on Offensive technologies (WOOT'11). USENIX Association, USA, 13.
- Ghaffarian et al. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. ACM Comput.
- Zhou et al. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Advances in Neural Information Processing Systems.

Vulnerability Discovery through Machine Learning supported Fuzzing I

- Consists of repeatedly testing an application with modified inputs with the goal of finding security vulnerabilities.
- The input grammar has to automatically generated and adapted.
- Idea: Use Machine Learning to guide the process of input generation.
- Give an overview on Fuzzing and the use of different ML techniques to support the process, find and discuss more related work.

Vulnerability Discovery through Machine Learning supported Fuzzing II

References

- Godefroid et al. Learn&fuzz: Machine learning for input fuzzing. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017.
- Yan, Guanhua, et al. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. 2017 IEEE Symposium on Privacy-Aware Computing (PAC). IEEE, 2017.
- Grieco et al. 2016. Toward Large-Scale Vulnerability Discovery using Machine Learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY ’16).

Discover Privacy Violations in Mobile Apps I

- Identify and list personally identifiable information that can be collected on Andoid and/or iOS
- Collect and evaluate methods for automated analysis of mobile apps to identify privacy concerns (e.g. Taint analysis)
- Describe counter-measures built into Android/iOS and/or provided by third party apps

Discover Privacy Violations in Mobile Apps II

References

- Mumtaz et al.: ”Critical review of static taint analysis of android applications for detecting information leakages”
- Enck et al.: ”TaintDroid. An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”
- Egele et al.: ”PiOS. Detecting Privacy Leaks in iOS Applications”
- Wang et al.: ”DroidContext. Identifying Malicious Mobile Privacy Leak Using Context”
- Wu et al.: ”Efficient Fingerprinting Based Android Device Identification With Zero Permission Identifiers”

Automatic Proof Generation for Software Security Verification I

- Recent advances in automated proof checking yielded performant tools for software verification
- Statically verifying that some (security) properties hold true for the program, leading to more secure and reliable software
- A good paper would contain a taxonomy of tools that have been used to automatically generate specifications for verifying programs. You would create a taxonomy to show the connection between
  - Verification tools like Smack, Prusti, Vcc, etc. and their categories (i.a. constraint solvers, symbolic execution engines, model checkers, etc.)
  - Underlying theorem provers and their languages / syntaxes (i.a., Z3, Coq, Lean, Isabell, etc.)
  - Intermediate verification languages (i.a., Viper, Boogie, Promela, etc.)
  - Classes of problems can be expressed (FOL, HOL, LTL, CTL, ...?)
- It would also contain answers to the following questions
  - How does the generation / translation from program code to verifier languages work?
  - Which *security* properties are verified in more applied literature (e.g., absense of exceptions, information leakage, data integrity, ...)?

Fraunhofer AISEC

Automatic Proof Generation for Software Security Verification II

References

- Leonardo de Moura and Nikolaj Bjorner. Satisfiability Modulo Theories: Introduction and Applications
- Astrauskas, Vytautas, et al. Leveraging rust types for modular specification and verification.
- Garzella, Jack J., et al. Leveraging Compiler Intermediate Representation for Multi-and Cross-Language Verification
- Qadeer, Shaz. Algorithmic verification of systems software using SMT solvers.
- Cohen, Ernie, et al. VCC: A practical system for verifying concurrent C.
- Oortwijn, Wytse, Dilian Gurov, and Marieke Huisman. Practical Abstractions for Automated Verification of Shared-Memory Concurrency.

Language-Aided Static Analysis I

- Language-based security techniques
  - = provable security guarantees by using the semantics of the underlying language
  - ⇒ can aid static verification of security properties in software
  - ⇒ shifts burden of analysis (e.g. writing pre/post-condition, security annotations) to developer who has usually more knowledge about code semantics
  - ⇒ uses the compiler to verify security properties, no need for external verifier
- For this work you should conduct research on papers, programming language specifications and web blogs, to compile a list of language features that aid static analysis of security properties
- Pay special attention to type systems and give an overview about which of their properties result in advantages for static security analysis
- To narrow the scope, you should focus on the most interesting well-known/industry-proven or major research languages that are *strongly-typed* e.g. Rust, Java, Kotlin, OCaml, Python, Haskell, Ada/Spark, Dafny

Language-Aided Static Analysis II

References

- Cardelli, Luca. "Type systems."
- Balasubramanian, Abhiram, et al. System programming in Rust: Beyond safety.
- Alexis Beingessner. The Pain Of Real Linear Types in Rust. `https://gankra.github.io/blah/linear-rust/`
- https://blog.adacore.com/spark-2014-flow-analysis
- Schoepe, Daniel. Flexible Information-flow Control.

Getting Started

- Understand the core concepts
  - Initial literature serves as a basis
  - Further research is expected
  - Check sources, follow-up work, and related publications
  - **When in doubt, reach out to your advisor**

- Structure your work
  - What contents do you plan to include in your paper? What is the contribution/the benefit for the reader?
  - Make an outline

- Further info on writing & preparing talks will follow

Q&A

Q&A

Fraunhofer
AISEC