The Integration of Multi-Color Taint-Analysis with Dynamic Symbolic Execution for Java Web Application Security Analysis

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurswissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Malte P. Mues

Dortmund
2023

Tag der mündlichen Prüfung: 02.03.2023
Dekan/Dekanin: Prof. Dr. Gernot A. Fink
Gutachter/Gutachterinnen:
Prof. Dr. Falk Howar
Prof. Dr. Dirk Beyer

Abstract

The view of IT security in today's software development processes is changing. While IT security used to be seen mainly as a risk that had to be managed during the operation of IT systems, a class of security weaknesses is seen today as measurable quality aspects of IT system implementations, e.g., the number of paths allowing SQL injection attacks. Current trends, such as DevSecOps pipelines, therefore establish security testing in the development process aiming to catch these security weaknesses before they make their way into production systems. At the same time, the analysis works differently than in functional testing, as security requirements are mostly universal and not project specific. Further, they measure the quality of the source code and not the function of the system. As a consequence, established testing strategies such as unit testing or integration testing are not applicable for security testing. Instead, a new category of tools is required in the software development process: IT security weakness analyzers. These tools scan the source code for security weaknesses independent of the functional aspects of the implementation. In general, such analyzers give stronger guarantees for the presence or absence of security weaknesses than functional testing strategies. In this thesis, I present a combination of dynamic symbolic execution and explicit dynamic multi-color taint analysis for the security analysis of Java web applications. Explicit dynamic taint analysis is an established monitoring technique that allows the precise detection of security weaknesses along a single program execution path, if any are present. Multi-color taint analysis implies that different properties defining diverse security weaknesses can be expressed at the same time in different taint colors and are analyzed in parallel during the execution of a program path. Each taint color analyzes its own security weakness and taint propagation can be tailored in specific sanitization points for this color. The downside of dynamic taint analysis is the single exploration of one path. Therefore, this technique requires a path generator component as counterpart that ensures all relevant paths are explored. Dynamic symbolic execution is appropriate here, as enumerating all reachable execution paths in a program is its established strength. The Jaint framework presented here combines these two techniques in a single tool. More specifically, the thesis looks into SMT meta-solving, extending dynamic symbolic execution on Java programs with string operations, and the configuration problem of multi-color taint analysis in greater detail to enable Jaint for the analysis of Java web applications. The evaluation demonstrates that the resulting framework is the best research tool on the OWASP Benchmark. One of the two dynamic symbolic execution engines that I worked on as part of the thesis has won gold in the Java track of SV-COMP 2022. The other demonstrates that it is possible to lift the implementation design from a research specific JVM to an industry grade JVM, paving the way for the future scaling of Jaint.

Keywords: Dynamic Symbolic Execution, Automated Software Testing, SMT Solving, IT Security, Multi-Color Taint Analysis, Software Engineering, Software Verification.

List of papers

I **Jaint: A framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of Java programs**

by Malte Mues, Till Schallau, and Falk Howar. In *Integrated formal methods: 16th International Conference*, IFM 2020, Lugano, Switzerland, 2020. (In this document cited as: [107])

II **JDart: portfolio solving, breadth-first search and SMT-Lib strings (competition contribution)**

by Malte Mues and Falk Howar. In *Tools and algorithms for the construction and analysis of systems: 27th International Conference*, TACAS 2021, Luxembourg City, Luxembourg, 2021. (In this document cited as: [103])

III **JDart: dynamic symbolic execution for Java bytecode (competition contribution)**

by Malte Mues and Falk Howar. In *Tools and algorithms for the construction and analysis of systems: 26th International Conference*, TACAS 2020, Dublin, Ireland, 2020. (In this document cited as: [102])

IV **Data-Driven Design and Evaluation of SMT Meta-Solving Strategies: Balancing Performance, Accuracy, and Cost**

by Malte Mues and Falk Howar. In *ASE 2021.* (In this document cited as: [100])

V **GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution)**

by Malte Mues and Falk Howar. In *Tools and algorithms for the construction and analysis of systems: 28th International Conference*, TACAS 2022, München, Germany, 2022. (In this document cited as: [101])

VI **SPouT: Symbolic Path Recording during Testing - a Concolic Executor for the JVM**

by Malte Mues, Falk Howar, and Simon Dierl. In *20th International Conference on Software Engineering and Formal Methods*, SEFM 2022, Berlin, Germany, 2022. (In this document cited as: [105])

VII **JConstraints: a library for working with logic expressions in Java**

by Falk Howar, Fadi Jabbour, and Malte Mues. In *Models, mindsets, meta: the what, the how, and the why not?: Essays dedicated to Bernhard Steffen on the occasion of his 60th birthday*, Springer, Cham, 2019. (In this document cited as: [77])

VIII **Thoughts about using constraint solvers in action**

by Malte Mues, Martin Fitzke, and Falk Howar. In *Electronic communications of the EASST 78*, Berlin, Germany, 2020. (In this document cited as: [98])

Comments on my participation

I **Jaint: A framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of Java programs**

I developed most of the ideas in close collaboration with Falk Howar. Till Schallau joined and adapted the DSL to the MPS implementation. The first prototype for combining multi-color dynamic taint analyses and dynamic symbolic execution is mostly my contribution. The final version distributed along with the paper is joint work with Till Schallau adapting his generator using the MPS framework. I co-authored all sections of the paper.

II **JDart: portfolio solving, breadth-first search and SMT-Lib strings (competition contribution)**

This work is our competition contribution with JDart at SV-COMP 2021. JDart become second winner in the Java competition. With smaller modifications I made to the binary for SV-COMP 2022, JDart becomes the winner of the Java competition. I have implemented the portfolio solving strategy and implemented all of the string analysis and CVC4 solver bindings. Falk Howar contributed the new implementation of the constraint tree allowing for the breadth-first search. I co-authored all sections of the paper.

III **JDart: dynamic symbolic execution for Java bytecode (competition contribution)**

This work is our competition contribution with JDart at SV-COMP 2020. JDart become third winner in the Java competition. The ideas presented have been discussed among all authors of the paper. I co-authored all sections of the paper and have done most of the preparation for participating with JDart at SV-COMP.

IV **Data-Driven Design and Evaluation of SMT Meta-Solving Strategies: Balancing Performance, Accuracy, and Cost**

All ideas presented in the paper have been developed in discussions among the authors. I have lead authored the paper in all sections except the parts around Feautre-based Solver Selection, which has been lead authored by Falk Howar and co-authored by myself. I have run most of the experiments, except those for solver selection with the sklearn library. These have been conducted by Falk Howar. The ASE commitee awarded an ACM Sigsoft Distinguished Paper Award for the paper.

V **GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution)**

All ideas presented in the paper have been developed in discussions among the authors. The implementation of GDart is joined work with Falk Howar. I lead authored the paper. GDart participated in the Java track of SV-COMP 2022. During the preparation, I participated actively in packaging the runscripts and setup the infrastructure for running the experiements before the submission.

VI **SPouT: Symbolic Path Recording during Testing - a Concolic Executor for the JVM**

All ideas presented in the paper have been developed in discussions among the authors. I lead authored Section 2.3 and co-authored all other sections. The implementation of SPouT is joined work with Falk Howar and part of the GDart tool.

VII **JConstraints: a library for working with logic expressions in Java**

I co-authored all sections of the paper. The FEAL idea was originally discussed between Falk Howar and Fadi Jabbour and the implementation and experiments are Fadi Jabbour's contribution. I contributed more active to the general description of the JConstraints library.

VIII **Thoughts about using constraint solvers in action**

This paper is a follow up to paper VII discussing various aspects of the integration of SMT solvers into tools. The ideas have been discussed among the authors. I lead authored the paper except for Section 4, which is based on of Martin Fitzke's bachelor thesis. I co-authored this section and have supervised Martin's thesis.

Other peer reviewed publications

- **GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution)**

  by Malte Mues and Falk Howar. In *Tools and algorithms for the construction and analysis of systems: 28th International Conference*, TACAS 2022, München, Germany, 2022.

- **The RERS challenge: towards controllable and scalable benchmark synthesis**

  by Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen. In *International Journal on Software Tools for Technoloy Transfer*, 23, Springer, 2021.

- **Do Away with the Frankensteinian Programs! A Proposal for a Genuine SE Education**

  by Simon Dierl, Falk Howar, Malte Mues, Stefan Naujokat, and Till Schallau. In *Third International Workshop on Software Engineering Education for the Next Generation*, SEENG 2021, Madrid, Spain, 2021.

- **Identification of spurious labels in machine learning data sets using N-version validation**

  by Malte Mues, Sebastian Gerard, and Falk Howar. In *IEEE 23rd International Conference on Intelligent Transportation Systems*, ITSC 2020, Rhodes, Greece, 2020. (In this document cited as: [99])

- **Teaching a project-based course at a safe distance: an experience report**

  by Malte Mues and Falk Howar. In *IEEE 32nd Conference on Software Engineering Education and Training*, CSEE&T 2020, München, Germany, 2020.

- **RERS 2019: combining synthesis with real-world models**

  by Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon Schiffelers, Harco Kuppens, and Frits W. Vaandrager. In *Tools and algorithms for the construction and analysis of systems: 25 years of TACAS: TOOLympics*, TACAS 2019, Prague, Czech Republic, 2019.

- **Generating component interfaces by integrating static and symbolic analysis, learning, and runtime monitoring**

  by Falk Howar, Dimitra Giannakopoulou, Malte Mues, and Jorge A. Navas. In *Leveraging applications of formal methods, verification and validation. Verification: 8th International Symposium,*, ISoLA 2018, Limassol, Cyprus, 2018.

- **RERS 2018: CTL, LTL, and reachability**

  by Marc Jasper, Malte Mues, Maximilian Schlüter, Bernhard Steffen, and Falk Howar. In *Leveraging applications of formal methods, verification and validation. Verification: 8th International Symposium,*, ISoLA 2018, Limassol, Cyprus, 2018.

- **Releasing the PSYCO: Using Symbolic Search in Interface Generation for Java**

  by Malte Mues, Falk Howar, Kasper Luckow, Temesghen Kahsai, and Zvonimir Rakamarić. In *ACM SIGSOFT Software Engineering Notes 41*, 6 (2017), 2017.

Other publications

- **Can We Trust Theorem Provers for Industiral AI?**

  by Falk Howar and Malte Mues. In *IEEE Software*, 38 (6), IEEE, 2021.
  (In this document cited as: [78])

Acknowledgements

First, I thank Falk Howar for giving me the chance to embark on this adventurous journey leading to this thesis. Apart from supervising me during this thesis project, I learned countless things from you about academia during our time at TU Clausthal and TU Dortmund. Your motivation and guidance kept me on track towards reaching my thesis goal over this time. I really enjoyed our countless and fruitful discussions during the last years.

Moreover, I have to thank Dirk Beyer and his team, especially Philipp Wendler and Martin Spießl, for running SV-COMP and sharing their tool stack with us. Without the VerifierCloud and BenchExec, the experiments in this thesis would have been much harder. SV-COMP helped significantly to keep up the motivation over the last years. Further, I thank Dirk for his feedback.

I thank the Automated Reasoning Group at AWS under the leadership of Byron Cook for three awesome internships in the US. The internships with you allowed me to learn more about the business side of automated verification tools. Further, I learned a lot from my colleagues during this time. I am also thankful for the Amazon Research Award granted to Falk Howar by the Automated Reasoning Group that supported my thesis financially during the last phase.

I thank Simon Dierl, Till Schallau, and Richard Stewing for the joint research projects during my time at TU Dortmund and the great collaboration. I am grateful for the discussions with Barbara Steffen on the discussion about the economics of software engineering during her time in our group.

I thank the students that worked closely with me during their theses. Especially, Martin Fitzke for discussing with me many questions related to running experiments on scale intensively, Oxana Warkentin for discussing normalization of SMT problems, and Robert Delhougne for sharing his first experiments with the GraalVM as fuzzer for Java Script. I also thank Sebastian Gerard for working with me on validating machine learning benchmarks.

Then I thank Bernhard Steffen and Mark Jasper from the chair for programming systems for inviting me to work with them on the RERS benchmark generation project.

Last but not least, I want to thank everyone from the chair for software engineering and chair for programming systems for a great atmosphere and many fruitful discussions in the coffee corner. I want to mention particularly Ute Joschko and Sevda Tarkun. Both supported me in all organizational questions and manage our back office in an excellent way. Further, I have to thank the team that has kept our servers up and running over the last years.

For the fun time and moral support during the last years, I thank Alnis Murtovi and Frederik Gossen.

Thank you all!

Data Availability Statement

All software components developed by myself as part of this thesis are publicly available. The experiments for this thesis are mainly run with the versions published along with the original paper. All experiments are run using BenchExec [22] and the VerifierCloud[1]. The JAINT framework is published on Zenodo in form of the JAINT artifact [106] accompaning Paper I [107]. The latest versions of GDART and JDART are available in their competition version in the SV-COMP 2022 [21] reproduction package. I used them for the experiments presented in this thesis. Further, the more recent development versions of these tools are found on GitHub in the space of the AQUA group[2]. The other verifiers used in the experiments are also taken from the SV-COMP 2022 reproduction package [21]. The reproduction package of this thesis [97] describes how to configure these binaries for the experiments in Table 6.1 and the change to JBMC for Table 6.2. For the applicable license to the different tools, I refer the interested reader to the license files included with the different used tools. For Paper IV [100], the reproduction package is published on Zenodo as well [104].

[1] We used a modified version from the original VerifierCloud Jar published at: `https://svn.sosy-lab.org/software/ivy/repository/org.sosy_lab/vcloud/`
The modifications adress thesis specific additions to the included BenchExec version that allow to run all tools used in this thesis.

[2] `https://github.com/tudo-aqua/`

Contents

List of Figures

Listings

List of Tables

1 Introduction

The world around us is transforming step by step into a software-defined environment, and during the twenty-first century this trend has not only continued but even speeded up. Gutenberg's invention of the letterpress was only 500 years ago. And the letterpress with movable letters is often called the most important invention of the millennium, as it made knowledge accessible to everyone at a low price. Today, stable and fast access to the internet has in most places replaced printed information distributed on paper, taking over the importance printing machines have possessed for the last 500 years. While the internet has had many positive effects and helped bridge social distancing during the corona pandemic, both data protection and attacks on personal data or critical infrastructure have become easier. To name some examples: In 2020, a cyber attack encrypted the IT systems at the university medical center of the University of Düsseldorf, forcing the hospital to go out of service for several days, even with its emergency care[1]. In May 2021, the Colonial Pipeline shut down after ransomware encrypted the billing system. Colonial Pipeline Co. paid 75 bitcoins for the decryption tool. The pipeline was offline for six days, impacting fuel supply in various East Coast areas in the US. In August 2021, T-Mobile lost the data of more than 40 million customers, including first name, last name, date of birth, and the SSN allowing an attacker to identify as this person in many other places[2]. This information was an excellent starting step for further social engineering attacks on individuals. While it affected only a small number of people compared with the Equifax data breach in 2017, the repeated breaching of data makes it harder for individuals to prevent identity theft. Equifax set up a program spanning at least seven years to help affected individuals recover from identity theft linked with the data breach[3].

The Changing View of IT Security. The examples described above demonstrate that securing modern software applications has become part of the product quality requirement and is no longer only a matter of risk prevention. Traditionally, IT security was seen as an aspect of risk management [60], depending on the protection goal for specific usage of the software, and it was often considered part of operating the software (cf. the very short sections related to security in the SWEBOK [30] compared to functional testing). In these circumstances it was possible to ignore security risks in

[1] `https://www.uniklinik-duesseldorf.de/ueber-uns/pressemitteilungen/detail/it-ausfall-an-der-uniklinik-duesseldorf` {last accessed: Februar 2022}

[2] `https://www.t-mobile.com/news/network/additional-information-regarding-2021-cyberattack-investigation` {last accessed: February 2022}

[3] `https://www.ftc.gov/enforcement/cases-proceedings/refunds/equifax-data-breach-settlement` {last accessed: February 2022}

the implementation of a software design, as the protection of the system was seen as an operational task.

However, IT security breaches have started to evolve from a potential threat to a significant financial risk. In the early 2000s, Garg et al. [68] working for Ernst & Young estimated that the risk of security breaches would be priced into IT companies by mid of 2005. Back then, they also highlighted that the real losses for a company around that time were ten times higher than estimated by the company itself. While the estimates in their study ranked at less than $2 million, real losses were already at that time around $17-28 million. Today, there is a better estimate of the damage potential of IT security breaches. Additionally, new regulatory guidelines such as GDPR in the EU have intensified the game around IT security, as high fines add to the damage caused by the breach. In October 2020, the Information Commissioner's Office (ICO), enforcing the GDPR law, fined British Airways £20 million for the breach of customer data of 400,000 customers[4]. The ICO explicitly argued that British Airways had not undertaken sufficient action to mitigate the attack. This fine did not cover the cost of fixing the breach itself or the (probable) costs of compensating customers.

So cyber security is a business risk leading to potentially large financial losses, and lawmakers set incentives to invest upfront in order to avoid these risks. This raises the question why cyber security is still a huge issue and not a solved problem by now. The answer is threefold: First, it is a very complex problem, and the situation evolves quickly. Second, there is a shortage of IT security specialists. Third, IT security is seen as a soft non-functional requirement and not a measurable quality aspect of an implementation.

Establishing IT security is a complex challenge. The complexity results from the many ways in which the security of a system can be compromised, and the evolution of new attack vectors over time—variants that require frequent reevaluation of security design. A system's IT security assessment evolves over a software product's lifetime. Selling the product does not end the task of securing it. The Common Weakness Enumeration (CWE) project[5] is aimed at standardizing software patterns that manifest a potential security weakness. Nevertheless, new vulnerabilities that exploit these known weakness patterns in current software are reported every year to the CVE list[6]. Apparently, we — as the software engineering community — are not very good at avoiding these weakness patterns in software. At the same time, the list of known weaknesses grows over time. As it is only possible to check the absence of known weaknesses from a software product and the list of known weaknesses updates over time, the decision on the absence of weaknesses is only valid for a certain period and needs to be renewed periodically. The Log4j security breach in 2021, for example, demonstrates that even the security of established software is not granted and that an existing weakness will eventually turn into an exploited vulnerability. Furthermore, from a business perspective, IT security requires ongoing investment in the product, while the development is often considered a project with a

[4] https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2020/10/ico-fines-british-airways-20m-for-data-breach-affecting-more-than-400-000-customers/ {last accessed: February 2022}

[5] https://cwe.mitre.org {last accessed: February 2022}

[6] cve.org {last accessed: February 2022}

clearly defined end.

The requirement for periodic reevaluation and security assessments leads to the second problem. Someone has to do these. (ISC)^2 estimates that 4.19 million cyber security professionals established and maintained IT security worldwide in 2021[7]. The same study estimates a gap of 2.72 million additional persons to defend organizations' critical assets effectively—–the complexity of software weakness detection and the shortage in the workforce influence each other. There is a shortage of professionals to keep up with the demand. However, as a cyber security professional needs thorough knowledge and understanding of security weaknesses and testing, it is difficult to train additional professionals quickly. As human labor does not cover the problem, automation can perhaps be called to the rescue (e.g., the automated reasoning group at Amazon[8], Project OneFuzz at Microsoft[9], or CodeQL on GitHub[10]). A very successful example of fuzzing is the american fuzzy lop (AFL) project developed and kick-started at Google[11]. *Fuzzing* is a semi-automated random security testing technique that successfully detects security weaknesses—–at least AFL does. Fuzzing seems to be sufficiently mature and cheap enough to be integrated into the new standards for automated security testing in modern software development processes: e.g., with the ISO/SAE 21434 (released in 2021), security fuzzing has become part of the standard development process for automotive software. Nevertheless, fuzzing, by design, only spots security weaknesses by chance. It is an automated way of guiding human attention and not the final answer to the problem with security weaknesses encoded in software. Fuzzing is a good first step toward security testing and thereby improving the security of existing IT implementations, but it does not guarantee the absence of security weaknesses and cannot, therefore, certify the security of products.

If automation of data processing becomes the defining aspect for establishing a reasonable profit margin, excluding IT security weaknesses in an implementation is no longer optional and has to become verifiable and measurable. Therefore, a system's functional correctness must be tightly coupled with assumptions on the security of its data. A messenger platform promising end-to-end encryption fails in its main function (secure communication) if an unencrypted data flow exists between the two ends. As a consequence, the question how to prove the security of systems is a driving motivation for the research presented in this thesis. In the near future, certifying the absence of implementation-related IT security weaknesses in the software development process will become the goal for any piece of software built to state-of-the-art standards.

A Taxonomy of Faults Leading to IT Security Vulnerabilities. Modern interconnected systems have security threats resulting from faults in implementation (e.g., SQL

[7] `https://www.isc2.org//-/media/ISC2/Research/2021/ISC2-Cybersecurity-Workforce-Study-2021.ashx` {last accessed: February 2022}

[8] `https://www.amazon.science/latest-news/how-awss-automated-reasoning-group-helps-make-aws-and-other-amazon-products-more-secure` {last accessed: February 2022}

[9] `https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/` {last accessed: February 2022}

[10] `https://codeql.github.com` {last accessed: February 2022}

[11] `https://github.com/google/AFL` {last accessed: February 2022}

injection attacks) and faults allowing abuse of interconnection (e.g., Denial-of-Service attacks). According to the fault taxonomy presented by Avizienis et al. [7] these are only two of three classes of faults. These authors distinguish physical faults as a third category apart from development and interaction faults. Physical faults cover IT security weaknesses resulting from hardware modification, but they are beyond the scope of this thesis, which focuses exclusively on software. Avizienis et al. [7] call an exploited fault that has been used to compromise the IT security of a system *active* and a not yet exploited fault *dormant*. In this thesis, I will follow the modern tradition and call a dormant fault a *security weakness*. An active fault is a *security vulnerability*.

I will call the development faults in an implementation *hard IT security weaknesses*, as their existence is verifiable during development, e.g., an SQL injection attack is always enabled by an implementation fault. Hence it is possible to prove the absence of exploitable hard weaknesses in an implementation. In contrast, I will call the other category *soft IT security weaknesses*, as these faults are not directly caused by the software itself: e.g., a Denial-of-Service attack uses a system according to its specification and does not exploit a weakness in the implementation; the security violation results solely from the aggressive volume of requests during the attack on the system, abusing the specified behavior of the system. Strategies for dealing with fraudulent requests are part of the design but will require countermeasures during operation. This thesis focuses on detecting or proving the absence of *hard* security weaknesses in the implementation, which is a quantifiable quality measurement, at least for weaknesses listed in the CWE database. More explicitly, I will focus here on securing the implementation of Java web applications. Operational aspects, such as securing the execution on tamper-proof hardware and preventing Denial-of-Service attacks, remain an IT security risk that requires countermeasures in operational strategy.

So let's look more into the hard security weaknesses of Java web applications. The Open Web Application Security Project (OWASP) cultivates a list of the top ten web application security concerns, the so-called OWASP Top Ten [143]. All of these are detectable during development. When work on this thesis started, injection weaknesses topped this list as the biggest risk factor in the 2017 version. In the newer 2021 version[13], however, they fall back to third place: broken access control and failing cryptography have become bigger issues. Nevertheless, as detecting injection weaknesses is still a widely-known problem and an appropriate example for the taint analysis domain, I will use it as the driving case for explaining the security analysis developed in this thesis. Listing 1.1 demonstrates such a possible injection weakness with an example for SQL injection taken from the OWASP Benchmark, a set of tasks for measuring the detection performance of security analysis tools for Java code. A web service handler reads untrusted data from the web in the form of a request object (cf. line 6 and line 13), processes this data (cf. line 16), and adds the data without proper sanitization to an SQL query (cf. line 20). The resulting SQL query string is: *INSERT INTO users (username, password) VALUES ('foo', '<param>')* where *<param>* is a placeholder for

[12] `https://github.com/OWASP-Benchmark/BenchmarkJava` {last accessed: March 2022}
[13] `https://owasp.org/Top10/` {last accessed: March 2022}

```
...
@WebServlet(value=”/sqli-00/BenchmarkTest00018”)
public class BenchmarkTest00018 extends HttpServlet {
  ...
  @Override
  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType(”text/html;charset=UTF-8”);
    String param = ””;
    java.util.Enumeration<String> headers =
      request.getHeaders(”Test18”);
    if (headers != null && headers.hasMoreElements()) {
      param = headers.nextElement(); // just grab first element
    }
    ...
    param = java.net.URLDecoder.decode(param, ”UTF-8”);


    String sql = ”INSERT INTO users (username, password)”
      + ”VALUES ('foo','” + param + ”')”;

    try {
      java.sql.Statement statement =
        org.owasp.benchmark.helpers.DatabaseHelper.getSqlStatement();
      int count = statement.executeUpdate( sql );
      org.owasp.benchmark.helpers.
        DatabaseHelper.outputUpdateComplete(sql, response);
    } catch (java.sql.SQLException e) {
      if (org.owasp.benchmark.helpers.DatabaseHelper.hideSQLErrors) {
        response.getWriter().println(
          ”Error processing request.”
        );
        return;
      }
      else throw new ServletException(e);
    }
  }
}
```

Listing 1.1 Example of an SQL Injection taken from the OWASP benchmark task 18[12]

the content of the *param* variable in the example. A malicious requester of this web service might inject arbitrary data to the SQL query by exploiting this weakness and adding, e.g., the new user ”Mallory” to the database. The request has to complete the actual value pair first before adding a new value to insert. An exploiting string is for example: *x'), ('Mallory','hack123.* If this string is rendered into the existing snippet, the resulting SQL query is: *INSERT INTO users (username, password) VALUES ('foo', 'x'), ('Mallory', 'hack123').* Most modern database systems create a second user ”Mallory” with the password ”hack123” while executing this query. The successful exploit does not merely change the password of “foo” but creates a new user. Consequently, an SQL injection can be used as a first step to prepare the system for a further attack.

Security weaknesses like this SQL injection are well known to developers, who are,

[14] https://xkcd.com/327/

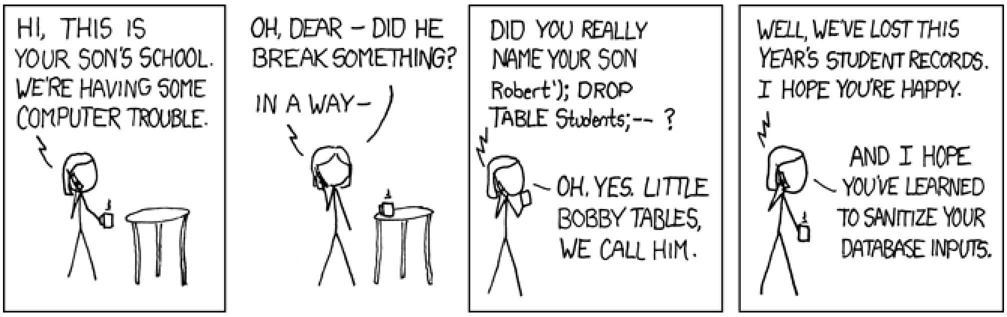

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 1.1: Exploits of a Mom by Randall Munroe[14]

however, notoriously bad in avoiding them, making it necessary to run automated security checks. SQL injection weaknesses have become part of pop culture as the comic in Figure 1.1 shows, and developers laugh about them. At the same time, as shown in a recent study by Braz et al. [32]], developers are unlikely during code reviews to find all injection weaknesses that result from improper input validation. They name inattention to security as one of the reasons why developers miss these weaknesses in the code review. History teaches us that these weaknesses will likely be exploited eventually, as the growing number of reported injection-related vulnerabilities in the CVE database demonstrates each year[15]. In consequence, the best solution is to automate the security audit and rerun it on every build. Automation makes it independent from human alertness and attention and will hopefully lead to securer software in future.

The integration of security into the software delivery pipeline as automated tests is an emerging trend built on top of the DevOps movement sometimes called *DevSecOps* [108][16]. Atlassian, a vendor of software development process management tools, defines DevSecOps as the future of security and recommends that you use at least one static analysis security testing (*SAST*) tool (e.g. [87, 136, 139]) and one dynamic analysis security testing (*DAST*) (e.g. [70, 73, 109, 111]) tool in your CI pipeline[17] to combine the best of both worlds. Mainly driven by Hdiv[18] and Contrast security[19] a third category of tools has been formed: interactive application security testing (*IAST*). IAST tools are often presented as the silver bullet for analyzers in DevSecOps pipelines and for protecting applications.

For explaining the challenges and problems of DAST, SAST, and IAST in more detail, we compare these tools in different dimensions: soundness, completeness, precision, and recall. *Soundness* describes the potential of an analysis to guarantee the absence of security weaknesses in a software program. This means an analysis is sound if the program

[15] Go to `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=injection` {last accessed: February 2022} for searching injection-related vulnerabilities in the CVE database.

[16] DevSecOps is sometimes also called SecDevOps or DevOpsSec, but I will refer to it as DevSecOps.

[17] `https://www.atlassian.com/continuous-delivery/principles/devsecops` {last accessed: February 2022}

[18] `https://hdivsecurity.com/owasp-benchmark` {last accessed: February 2022}

[19] `https://www.contrastsecurity.com/owasp-benchmark`{last accessed: February 2022}

has no security weaknesses when the analysis claims that there are none. *Completness* describes the property that if there is a security weakness in the program, the analysis is capable of finding it. Achieving soundness is often easier than achieving completeness. Soundness allows an analysis to find a security weakness that turns out to be a false alarm. Completeness requires that any security weakness that is found is a true security weakness. The ideal security analysis is both sound *and* complete, but this is rarely achieved in practice. Soundness prevents an analysis from falsely claiming that an insecure system is secure. Of the two, soundness is therefore seen as the more important. *Precision* measures how many of the predicted security weaknesses are true alarms and not false positives. A precise analysis reports only true positives. The final dimension, *recall*, measures how many of the existing security weaknesses have been found by an analysis. An analysis with perfect recall finds every security weakness. Let's look, then, into the characteristics of the three flavors of analyses regarding these dimensions.

Static Analysis Security Testing. SAST tools analyze the source code without executing the code. Consider the example in Listing 1.1. The computation of a control flow graph and computing the taint flow within the graph (e.g., the TAJ framework [139]) is one possibility for implementing a static analysis. An abstract interpretation engine (cf. for a survey Cousot and Cousot [49]) is another common way of implementing static analysis. The static analysis can only build models for the code available for analysis. Suppose the code under analysis is solely the code snippet in Listing 1.1. In that case, the static analysis has to make approximations for unknown code, e.g., the implementation of the *getHeaders()* function used in Line 11. Depending on the scope of the approximation model used, the resulting analysis will underapproximate or overapproximate the real program behavior at this point. This is the case, because the result value of the missing function directly influences the executable branches in the control flow graph. By design, this leads to the problem that the approximation is wrong and the resulting analysis is unsound or incomplete. Therefore, SAST tools often choose the sound solution of overapproximating the behavior. Consequently, they raise false positive alarms requiring manual inspection after the analysis terminates. This makes the analysis incomplete. . Underappoximations could be more precise (up to the point where they are complete), but also miss potential security weaknesses (making the analysis unsound). As missing a potential vulnerability is considered worse than raising a false alarm, static analyses usually overapproximate the implementation, optimizing for soundness at the cost of completeness. On the bright side, these tools are typically very fast. As they overapproximate the concrete behavior in various places, they can often analyze implementations that are still work in progress. This allows them to be used in DevSecOps pipelines, as they are cheap to execute and applicable in early design stages. Furthermore, filter rules eliminate known false positives from the output after manual inspection, reducing the number of false positives. Therefore, SAST tools are proven to deliver value in practice. Nevertheless, lowering the false positive rates is an ongoing challenge to make their application cheaper and suitable for software development teams that do not have access to IT security specialists for post-processing the results.

Dynamic Analysis Security Testing. DAST tools summarize the group of black box dynamic testing tools. Examples of techniques in this category are learning-based testing (e.g., Fiterău-Broştean and Howar [65]), black box fuzzing (e.g., the previously mentioned AFL project), and automated penetration testing (e.g., the OWASP ZAP project[20]). These tools have in common that they analyze real observable behavior without knowing any of the execution internals. This requires running the analysis target; therefore, it is often impossible to analyze only parts of the system without using complex mock setups similar to those required for integration testing. Moreover, these techniques execute code and hence will eventually also execute a vulnerability. Therefore, the system under test has to be run in a sandbox environment that allows a safe reset even after a successful attack. The results will be precise as the analysis can test for vulnerabilities using different strategies executing the tool. It is always possible that a tool misses a vulnerability, so the analysis is unsound.

Interactive Application Security Testing. The industry, led by Contrast security and Hdiv, drives the branding of IAST tools as the next generation of DAST and SAST tools. They are typically implemented within a tool's runtime, allowing access to any information available during execution. Passive IAST tools work as a monitor preventing access during the execution and sometimes serve as a web application firewall (e.g., TokDoc [85], or commercial products by Akamai [21]). Active tools allow precise monitoring during execution but require some input driver that is combined with the precise monitors. The dynamic component guarantees precise alarms, making it a complete analysis. Soundness depends entirely on the input generator. IAST tools have to overcome the same challenges as DAST tools, because both techniques execute programs. From an academic perspective, IAST tools are white-box DAST tools and techniques such as dynamic tainting (e.g., Livshits [86], or Song et al. [135]) and directed automated random testing (cf. Godefroid [69]) fall in this area.

1.1 Research problem addressed in this thesis

We have seen so far that IT security weaknesses may be encoded in the source code of a program, and I have presented the need, from a business perspective, for a software development process that certifies the absence of such weaknesses. Given the current shortage of trained IT security professionals, this process requires a high level of automation. Paired with the consideration of existing tools, this leads to the motivating vision of this thesis:

I envision a development process, supported by automated tools, that prevents IT security breaches caused by programming errors. The process does this by certifying software products that are free from known security weaknesses.

SAST, DAST, and IAST tools work in this direction, but all of them currently have weaknesses hindering full achievement of the goal. Given the current state of the art,

[20] `https://www.zaproxy.org` {last accessed: March 2022}

[21] `https://www.akamai.com/de/products/app-and-api-protector` {last accessed: February 2022}

there are two distinct approaches for advancing significantly toward the core certification aspect of the research vision: combining SAST and DAST tools, and enhancing the capabilities of IAST tools.

Option 1: Combining SAST and DAST tools. Because overapproximations leads to false positives in SAST tools, they are not precise enough to certify the absence of IT security weaknesses for programs in the general case. In theory, there is a sweet spot allowing the creation of SAST tools with an overapproximation of the real problem that does not report false positives in some of the use cases. Today's tools are not at this sweet spot, and it is hard to find the universal overapproximation that allows this on all potential programs. Nevertheless, as a result of the overapproximations, SAST tools are sound. In contrast, DAST tools are precise but miss too many examples, so they do not guarantee soundness in their state today. Combining and balancing the soundness of SAST tools with the precision of DAST tools is one possible research direction toward the research vision. The required work has to focus on solving the integration challenges between the designs of SAST and DAST tools allowing an efficient exchange of information between the different tools.

Option 2: Enhancing the capabilities of IAST tools. IAST tools have demonstrated their efficiency in monitoring execution along a concrete path and can guarantee to find potential security weaknesses on the monitored path. They deliver an excellent precision for certifying the absence of security weaknesses, as false positives will not be an issue. But in the current state of the art, IAST tools do not give guarantees on the absence of security weaknesses in the complete program, as they cannot guarantee the execution of all feasible paths in the program, making them unsound. Therefore, the required work involved for this research approach lies in pairing IAST tools with a required counterpart that allows guarantees on the soundness of the analysis.

Other Aspects of the Research Vision. Choosing either option 1 or option 2 for building the strategic certification core of the tool does not answer any of the questions regarding the integration of the solution in the development process, the cost-efficiency of the process, the presentation of security weaknesses to the user (e.g., a problem tackled recently for SAST by Luo [90]), or the validation of the certificates (e.g., the SV-COMP community invests significant effort in the validation of verification witnesses, a form of certificate validation [19]). Each of these areas implies sufficient questions and material for independent research projects contributing to the research vision. I have chosen in this thesis to focus on the core certification tool, as finding a reliable method that either provides guarantees on the absence of security weaknesses or detects them precisely is a promising first step toward realization of my research vision. Answering the other aspects mentioned will represent the next steps in integrating the proposed tool into the development process, thus closing the gap between my thesis and my research vision.

Chosen Research Question and Solution Proposal. Between the two presented options, I decided to invest in exploring the second option. Tightening the focus further, I have chosen the JVM as a demonstration target because it is a modern virtual machine with restricted memory access. The unrestricted memory access in assembly and C makes it significantly harder to certify security (e.g., see the argument proposed by Slowinska and Bos [134] regarding pointer tainting). Certain guarantees can only be given if assumptions on the runtime are made. Livshits [86] also argues in favor of security analyses in the context of a managed runtime instead of discussing security on the computation model of a modern CPU. Given that the usage of abstraction increases in today's software ecosystems, it is valid from my point of view to address required assumptions on the runtime to achieve the research vision. In this thesis, my assumption relies on the abstraction of the runtime provided by a JVM. For presentation in this thesis, I focus on the security weaknesses in JAVA web applications running on top of the JVM. However, the approach can be adapted to any other JAVA software.

As described above, the software under analysis is not only checked once but has to be recertified after every newly discovered security weakness or any source code change. Following the DevSecOps spirit, the answer to this question must be a method that can be implemented as a tool with the potential to run most proofs in future automatically.

All these considerations combine to form the central research question of my thesis

How is it possible to prove the absence of a security weakness in the source code of a JAVA web application by an automated tool, and if this is not possible, as there is a security weakness in the code, how can this weakness be detected precisely by the tool?

Luckily, a well-researched monitoring technique already exists that can detect weaknesses in the dynamic execution of a single path: *dynamic multi-color taint analysis.* I will explain the details of dynamic taint analysis and how it works in Chapter 5. Its most important capability is detecting potentially harmful data flow on the currently executed path. Dynamic tainting detects the malicious data flow as shown in the web servlet example presented in Listing 1.1. The downside is that dynamic taint analysis works only on the current path of concrete execution. Hence, it needs a driver technique that executes taint analysis on every possible path in a web servlet that is part of the web application under analysis. *Dynamic symbolic execution* is a source code analysis technique that does exactly this: enumerating all paths reachable within a program from a given entry point. Combining both techniques leads to the solution to the question presented in this thesis:

The combination of dynamic multi-color taint analysis with dynamic symbolic execution for the security analysis of JAVA web applications.

Dynamic multi-color taint analysis generates a precise counter-example that detects a security weakness, if proof of the absence is impossible. This counter-example points to the weakness, allowing an easy fix of the software under analysis. The high precision of the results that describes the weakness makes this easier. Further, making a verdict on the absence of weaknesses requires soundness in the analysis. An unsound analysis cannot give the guarantees required for proving the absence of weaknesses in the source code.

Constructing the suggested solution in this thesis combines knowledge from three domains closely entangled in the analysis but with independent bodies of research in the past: dynamic taint analysis, dynamic symbolic execution, and SMT solving. This thesis combines all three domains in the design of the analysis framework JAINT [107]. It is a framework for analyzing security weaknesses in JAVA web applications and can prove the absence of weaknesses if the analysis terminates. Otherwise, JAINT detects security weaknesses precisely. For building JAINT, I look in further detail for answers to the following three questions, which contribute to the main research question as sketched out below each of them:

DQ1: How is it possible to implement a precise and generic dynamic taint tracking architecture for explicit tainting that is configurable for different analyses and allows sanitization?

Configurable Multi-Color Taint Analysis. Dynamic taint analysis is well understood as a monitoring technique for detecting security weaknesses. A major technical challenge for the analysis application is tailoring it to the context of the usecase by configuration. For this thesis, I integrate an abstract taint analysis into the runtime that allows configuration of the analysis later for different security weaknesses rather than hard-coding the configuration. The configuration supports explicit sanitization and the parallel execution of multiple configured analyses with a unique taint color each. The taint tracking must be precise so that violations of the taint policy result in precise counter examples.

DQ2: How is it possible to combine dynamic multi-color taint analysis with dynamic symbolic execution for the analysis of JAVA web applications?

Parallel Execution of Dynamic Symbolic Execution and Multi-Color Taint Analysis. In its core idea, dynamic symbolic execution enumerates all possible paths of a program, and the multi-color taint analysis scans for security vulnerabilities along these paths. In the context of JAINT, these analyses are independent and are implemented in the same runtime. As tainting is precise, it will detect security vulnerabilities along a path. If the symbolic search terminates, all reachable paths of the program under analysis are enumerated. Completing this process without any taint property violations implies that they are not present in the reachable state space of the program. This way, the approach proves the absence of security vulnerabilities.

Dynamic symbolic execution is a powerful technique and has, in the existing literature, proven itself on JAVA programs working mainly with primitives (e.g., the work by Luckow et al. [89]). At the same time, it is well known that analyzing the paths of web applications requires support for string operations in the analysis (cf., [35, 41, 120, 132]). Therefore, extending the capabilities of existing dynamic symbolic execution engines for a program with strings is the main challenge tackled in this area, enabling both analyses in combination.

DQ3: How precise is the analysis of JAINT and what implementation decisions influence JAINT's recall?

SMT Meta-Solving Strategies. Dynamic symbolic execution relies critically in the backend on SMT solvers for symbolic reasoning. Consequently, the performance of the SMT solving layer directly influences how often the symbolic search space is completely explored and whether Jaint delivers the promised guarantees regarding recall and precision. Integrating reasoning on string operations in the symbolic problems involves a new category of SMT solvers: string theory solvers. The string theory of SMT-Lib is comparably complex and has led to a bouquet of string theory solvers that focus on different aspects of string theory. Therefore, I will look into SMT meta-solving strategies that combine multiple string solvers. As a meta-solving strategy combines multiple solvers, it allows the dynamic symbolic execution engine's performance to be decoupled from a single solver's performance.

The work presented in answer to these three driving questions makes it possible to build Jaint and contributes to the journey of scaling the framework for the analysis of real-world software. As we argue in Paper I [107], the analysis is precise as tainting does not find false positives, and for the boundaries of the symbolic search space sound, if the search terminates. Tailoring the search space adequately is a challenge in itself, not discussed in detail in this thesis, but I will describe how the soundness of the analysis links to the search space design. In the following pages, I will sketch the contributions made in these three areas, summing up the state of the art first and then going on to discuss the contributions and thus leading to an answer. Addressing these partially independent challenges allows construction of a more powerful and scalable analysis engine at the core of the Jaint framework. The framework forms the overarching context combining all contributions in one answer to the main research question.

1.1.1 Dynamic Taint Analysis

Various papers in the existing literature describe the benefits of dynamic taint analysis in general and for security as a specialty. For this thesis, I cannot discuss all of them. Instead, I will present a selection of relevant previous work influencing this thesis.

The area of taint analysis is split into two main research directions regarding tool building: dynamic taint analysis (e.g [45, 72, 73, 135]) and static taint analysis (e.g. [87, 139]). Additionally, some papers discuss the underlying theoretical foundations of taint analysis (e.g. [12, 86, 124, 126, 127, 147]) and how this method might be formalized. In the Jaint [107] framework, we have presented how dynamic taint analysis can be combined with dynamic symbolic execution. Therefore, I will focus on discussing related work on dynamic taint engines and the general theory about taint analysis. I will start with the use cases of taint analysis before briefly discussing six specific challenges.

Application of Taint Analysis. Tainting, independent of the implementation as static or dynamic analysis, is used for different application areas. There are three major areas in which dynamic taint analysis is used: stack protection in assembly based languages (e.g. [45, 67, 135]), preventing injection attack in web applications (e.g [72, 86, 109, 111]), and tracking information flow (e.g. [5, 48, 64, 74, 124]). Livshits [86] mentioned this separation for the first time in his technical report about taint tracking in managed runtimes, but this separation has not established itself in academia as a categorization for taint tracking work. Consequently, the challenges are not well defined for these three areas, and they get occasionally mixed up as all of them relate to taint tracking.

```
output=0;
if (secret==1000) {
    ...
    output=1;
}
send(output);
```

Listing 1.2: Example of strong control dependence taken from [12].

Challenges for Taint Analysis. I found six challenges for taint analysis considered and discussed in the literature. Two of them do not apply for taint analysis of JAVA programs or the given usecase, one is left as future work for JAINT, and for the remaining three, I will present the strategy to deal with them in this thesis:

1. *Implicit Flow vs. Explicit Flow.* Taint analysis marks data values and tracks the flow of data from a sink to a source. The literature distinguishes mainly between two types of taint: explicit and implicit taint flow (cf. Denning and Denning [57] or Sabelfeld and Meyers [124]). Explicit taint analysis tracks the actual data flow as the taint mark is linked to the data value itself and passed along with any assignment. An explicit flow violation occurs if, e.g., a password is written directly into a logging file. In contrast, indirect information leakage is gained by observing the program during execution and learning about secret data from observation. Taint analysis techniques targeting weaknesses using the indirect information leakage are called implicit flow analyses. The following paragraph briefly explains implicit flow in more detail, as it is often linked with information flow and is specified as a core challenge for information flow analysis in general. For this thesis, I will focus strictly on explicit taint flow, as it is the more common case in the context of injection attacks. Implicit flow challenges are, then, beyond the scope of this thesis, nor are they at the moment supported in JAINT.

   The implicit flow problem describes an information leak from a program because an attacker controls either a control flow or a data variable and uses this control to gain information. Listing 1.2 shows an example made by Bao et al. [12] for strong control dependency, a specialized form of implicit flow. This concrete example shows the implicit flow between the secret variable and the output variable. If attackers know the control flow, they can recreate the values from the secret variable observing only the system output across multiple runs. This information leakage across multiple executions in consequence of control structures is called implicit flow. The strong control flow as a form of implicit flow leaks the precise secret value over the control structure, while the weak control flow only establishes boundaries for the interval restricting the domain of the secrete value. Bao et al. [12] discuss the specialization of strong and weak control flow in greater detail. Moreover, they describe its effect on dynamic taint analysis and start to define strong and weak control flow dependencies for data flow. Independently of Bao et al. [12], The implicit flow problem has been well known for nearly two decades. Sabelfeld and Meyers [124] contributed significantly to the formal understanding of this problem.

2. *Pointer analysis.* Substantial parts of the previous work related to dynamic taint analysis have been done for the C, C++, and Assembly language stack. These languages always have to deal with pointers to memory addresses in the language. It is constantly necessary to track whether the address is tainted or the value written in the memory cell carries the taint. This tracking leads in different situations to over- or undertainting depending on how it is realized (c.f. tainting in the SimpIL language by Schwartz et al. [127]). Java does not possess pointers as the memory access is more strongly controlled than in other lower-level languages. In consequence, the discussion does not apply here. Nevertheless, I will briefly highlight the importance of the use case for judging the impact of pointer analysis on the suitability of taint analysis.

   The impact of pointer tainting on the usefulness of the analysis depends on the use case for the analysis. Techniques using tainting for tracking information flow and detecting potential malware cannot overcome the pointer analysis problem in a fulfilling way [39, 134]. In contrast, Dalton et al. [52] defend the efficiency of pointer tainting for stack protection against other less efficient use cases presented by Slowinska and Bos [134]. This discussion already shows how important the domain is for judging whether pointer tainting is a problem for target analysis or not. As explained in the previous paragraph, for analyzing JVM applications pointer tainting does not apply because of JVM's memory architecture.

3. *Undertainting.* Occasionally, a taint analysis does not propagate the taint marks in places where they should be propagated for a sound analysis. In consequence, the taint analysis misses a security weakness due to underapproximation of the real taint flow. This fault in the analysis is called *undertainting.* There are various root causes leading to undertainting. One cause is wrong assumptions in the analysis specification: for example, sanitization functions that do not sanitize the input as expected perform undertainting. Depending on the property that is the target of the analysis, missing implicit flow tainting may be a reason for undertainting. Moreover, increasing the precision of a taint analysis is a valid design goal for applying undertainting by choice. Some of the solutions for more precise pointer analysis, e.g., use undertainting to avoid tainting everything. A risk for undertainting in managed runtimes lies in models used to approximate the behavior of code that is abstracted away from the runtime. For example, for a Java program, it is very hard to tell whether a file is a link in the file system pointing to another protected file or not. As a link points potentially to an arbitrary file, an overapproximating model would taint any file that is a potential target of the link. To avoid spreading the taint marks across the complete file system, the undertainting marks only the filename of the symbolic link and not any potential target. This misses the requirement that the link target should have been tainted as well. The problem is comparable to the problem of tainting pointers. Therefore, it is hard to model data flow across the file system appropriately. Dealing with models for system resources, e.g., the file system, and defining the right tainting strategy is left as future work for the Jaint framework and is not addressed in this thesis.

4. *Overtainting and Implicit Sanitization.* Overapproximations used in taint analyses allow the tainting of data is tainted that cannot be reached by the data flow in a real run. Overapproximation helps to reduce the state spaces, sometimes leading to soundness, but also generates false positives, thus harming completeness.

   Sanitization removes taint marks from data values. It is the most important countermeasure to stop overtainting in systems. Sanitization exists in two forms: implicit sanitization and explicit sanitization. Implicit sanitization occurs if math functions compute constant values that cancel out the data flow. An example of implicit sanitization given by Schwartz et al. [127] is the xor operation in x86 assembly: $b = a \oplus a$. Taint flow has to stop on such byte codes for constant results. I will discuss how JAINT deals with implicit sanitization in Section 5.1. Explicit sanitization occurs if a function checks a parameter for a valid input domain. Determining the explicit sanitization functions for a program is a nontrivial task. Some work has been conducted on automatically identifying sanitization functions, e.g. [11, 76]. Nevertheless, the developer often has to name these explicitly if the framework supports sanitization. Therefore a human still needs to provide input to prevent overtainting. JAINT's configuration language (cf. Paper I [106]) supports the definition of sanitization methods for each taint color, as will be discussed in Section 5.3. The configuration names explicit sanitization points in a humanly readable format.

5. *Language Boundaries and the Right Place for Tracking.* Taint analysis is implemented in various places across the software stack. The right choice for the implementation strategy depends on the target language and purpose. Today, the race between software-defined taint tracking and hardware-accelerated taint tracking seems to have been called. Yet given the ubiquity of cloud infrastructure, allowing interception of arbitrary virtualization levels, deciding the right position and way to integrate taint tracking into the software stack is still an unsolved issue. For JAINT [107], we decided to instrument the JVM because it works and allows interception of the JAVA program execution in all relevant parts for the JAINT framework. Dalton et al. [50] claimed that instrumenting the interpreter has the disadvantage that code behind the JNI requires custom wrappers that establish security over the JNI boundary. In addition, they mention that multithreaded executables cannot be analyzed this way. Instead, their tool Raksha [50] has been realized on FPGA boards as a special hardware implementation simulating SPARC architecture. The tool tracks taint flow on the assembly level and has achieved notable results [51]. However, building custom hardware, even on an FPGA board, is a significant task. Moreover, today's hardware landscapes tend to diversify with the appearance of new ARM-based SoCs, for example, the M1 chip designed by Apple. Therefore, instrumenting the JVM seems more viable for JAINT than simulating the CPU as FPGA and has worked well so far.

   Other approaches focus on instrumenting the language interpreter, the byte code, and a virtualized machine. The only language that features it as an official part of the environment so far is Perl [4]. PHP supports a taint mode in the environment that can be enabled. Both implement taint analysis in the interpreter. Other attempts are implementations in the form of a library, e.g. [46], in the hardware architecture, e.g. [50], using virtualization support, e.g. [53, 75], in the runtime, e.g. as we have done for JAINT [107], and by instrumenting the byte code, e.g. [67, 72, 135]. On the other hand, the downside of whole system analysis is the general overhead, which has been decreased in recent years by improving tracking techniques, but is still measurable (c.f. [67, 94, 95]). In most cases, dynamic tainting is used for online analysis while running the program, e.g. [111]. In these cases, the overhead is important, as it impacts the productivity of the software with the given hardware resources.

6. *Time of Detection vs. Time of Attack.* The last problem presented in this thesis is time of detection vs. time of attack. JAINT's goal is the detection of weaknesses before deployment so that they are always detected before they are exploited. I will discuss this in more detail in Section 5.2 but will present the general problem briefly now. If an attack is detected while the program is running or after the fact, it might be already too late to prevent data loss or service agreement violations (c.f. [127]). Stopping a program's execution is only possible if the taint analysis can interact with the executing environment (e.g., [111]). If the taint analysis is a passive component detecting only the attack or works offline as in the Straight-Taint [94] framework, the detection can only happen as a post-mortem analysis. The remaining question is how to integrate the taint analysis into the development process so that it can prevent attacks. BitBlaze [135] has a combination of dynamic tainting with symbolic execution that allows analysis of the program during a test phase. The author does not address the problem of integrating BitBlaze into the software development process so that it detects security weaknesses before they are exploited. The goal remains, therefore, to detect weaknesses and vulnerabilities before they are used in an attack.

Contributions of this Thesis regarding Taint Analysis. The analysis engine presented here is designed to find security weaknesses in JAVA source code implementing the backend of a JAVA web application as described in the JAINT paper [107] (Paper I). It is the first analysis that combines dynamic symbolic execution with dynamic multi-color taint analysis for JAVA web applications. The closest existing approach is BitBlaze [135] for x86 assembly, as it also has some kind of symbolic execution engine that partially drives the taint analysis. However, as the x86 memory access is less restricted than in the JVM, BitBlaze has to deal with many problems that are not relevant for JAVA, mainly resulting from pointer analysis.

JAINT splits the taint engine in the JVM from its configuration for a particular analysis focusing on usability. Such a split has also been proposed by Livshits et al. [87] using PQL [92]. PQL's design allows more complex analysis of programs in their static analysis engine used to evaluate PQL queries. Consequently, PQL is more expressive than JAINT's configuration language, but this expressiveness comes at the cost of complexity. JAINT uses a domain-specific language designed only to configure the multi-color taint analysis. As it is a multi-color analysis, it is possible to define interdependent analyses in the DSL and run them all simultaneously. This approach follows the consideration of Metaprogramming as presented by Hunt and Thompson, especially the ideas summarized as ”Tip 38: Put Abstraction in Code, Details in Metadata” [79, p. 145].

The analysis presented here does not track implicit flow. So, it is incomplete for weaknesses that might exploit implicit flow. Moreover, the original JAINT [107] paper does not mention the Time of Detection vs. Time of Attack problem. While JAINT still supports explicit sanitization, I have reevaluated the decisions in this thesis compared to the original paper regarding implicit sanitization. Chapter 5 discusses all these ideas in greater detail. In what follows, I will present contributions made in the area of scaling and improving the dynamic symbolic execution component used in JAINT.

1.1.2 Dynamic Symbolic Execution

Dynamic symbolic execution is used for path enumeration in the JAINT framework and directly affects the soundness of the analysis. If dynamic symbolic execution misses a branch it is supposed to enumerate, JAINT might claim a program as secure that is not secure; therefore it is unsound. It is closely related to the more active area of symbolic execution introduced by King [82] in the 1970s and is a very active research area. Therefore, I cannot give an overview of every paper and different trends and challenges for the symbolic execution of different programming languages (But I refer the interested reader to the survey by Cadar and Sen [38] or more recently by Baldoni et al. [10]). Instead, I will focus on the remaining challenges for the symbolic execution of JAVA programs, together with a couple of milestones that seriously influenced the development of JDART as part of this thesis, leading to the participation at SV-COMP 2020 [102] (Paper III) and SV-COMP 2021 [103] (Paper II). Of course, the same consideration influenced GDART [101] and the design of SPOUT, GDART’s concolic executor [105].

State of the Art. From a theoretical viewpoint, symbolic execution closely relates to model checking (for more details about model checking see the handbook by Clark et al. [44]) and, as such, suffers from the state explosion problem in the same way as model checking. The *state explosion problem* in model checking describes the problem that different states with multiple possible transitions lead to an exponential growth of new reachable states in every new iteration of the transition system, making it impossible to model all of them precisely for model checking. Moreover, model checkers usually describe the global system state. If the available memory for the system state is large, this state space quickly becomes too large to describe in its entirety (cf. Chapter 1 of Clark et al. [44]). Over time, much work has been invested in more efficient symbolic encodings, reducing the impact of the state space explosion on model checking to enable reasoning on more complex state spaces [24, 36, 71]. The landmarks are using symbolic encodings [36], using SAT solvers instead of binary decision diagrams [24], and later using the powerful SMT solvers (e.g. [71]). Working with logic encoding allowed the addition of safe overapproximations that interpolate away irrelevant parts of the state space regarding the analysis property [93]. Jaffar et al. [80] have demonstrated how to combine interpolation with dynamic symbolic execution to reduce the state space that requires exploration during a dynamic symbolic execution run. Avgerinos et al. [6] combined dynamic and static symbolic execution to leverage the strengths of both techniques in a single tool. This starts running as a dynamic symbolic executor but switches at branching points to static symbolic execution whenever it is possible to summarize the effect of the branching decision symbolically. The symbolic summaries avoid forking the dynamic symbolic execution into all possible branches, reducing the state space that needs exploration. Unaffected by this progress in the theory, however, the three major challenges for applying dynamic symbolic execution in practice remain reasoning about unbounded heap data structures, path explosion, and expressiveness in constraint solving as pointed out by Păsăreanu et al. [119].

On the tool side, symbolic execution engines evolved from dynamic symbolic execution engines such as CUTE [130] that were developed as better unit test engines. CUTE explores all the execution paths of a program, consuming inputs by executing a path concretely and recording a symbolic representation that describes the constraints for branch decisions during the run. Later, it negates some of the constraints and passes them on to a constraint solver that computes new driving input values for a reexecution of the program under test, taking another branch. However, CUTE was written before SMT solvers emerged and therefore handles constraint solving in its own backend. Some parts of the constraints may be replaced with concrete values, resulting in a relaxation of the problem, which then becomes solvable. It is an early demonstration of the potential of this technology, which highlights the tight interweaving of success in the analysis enabled by success in the area of constraint solving.

Around the same time DART [69] appeared. In its core idea DART works similarly to CUTE, except that it detects the interface of the program under test automatically and generates the required drivers as a service for the user. Generating a driver method that executes the software snippet under test in the desired way has also been reported as an issue in dynamic taint analysis. Hence, this is a joint problem for both analysis approaches. Godefroid et al. [69] show how dynamic execution overcomes situations where the constraint solver cannot solve the path condition by using values obtained from concrete execution at branching points. Symbolic execution alone is unable to reason about software in such situations, as symbolic analysis cannot make any decision without a constraint solver.

With the introduction of more powerful SMT solvers (led in 2008 by Z3 [56]), the research area shifted back to pure symbolic execution. One of the popular symbolic execution engines for C is KLEE [37]. KLEE has successfully analyzed major Linux tools. The applicability of symbolic execution shifted from the problem of solving constraints and executing the program at all to the area of dealing with system calls and library calls that are out of scope for the current analysis. KLEE solved this issue by providing hand-coded symbolic models for different system calls.

For the java world, SYMBOLIC PATHFINDER [118] (SPF) is the leading symbolic execution engine. A few years later, Luckow et al. introduced JDART [89] as a dynamic symbolic execution engine sharing JAVA PATHFINDER [142] as the instrumented JVM with SPF. Using SPF, Redelinghuys et al. [120] discussed different approaches for integrating string operations into the symbolic execution of programs. They present two approaches for solving string constraints in a symbolic encoding: automatons and bitvector encoding. They also represent string operations in a newly introduced intermediate data structure called a *string graph*: a hypergraph that uses integer and string variables in the vertices and describes how they are connected by an operation at the edges. Apart from variables, vertices can also represent constants. These string graphs are transformed into the input for either the automaton based solver or the bitvector based solver. The authors concluded that neither of these two approaches is preferential over the other for symbolic execution with string constraints. Instead, they concluded that the integration between integer and string constraints in the solving procedure would be the most important feature for using string constraints in symbolic execution Java Ranger [133], a Java program analysis tool built on top of SPF, implemented vertitesting for Java and has shown that it is a powerful technique, but Java Ranger has limited support for string operations. Overall, a good theoretical understanding in this area leaves the question of how to leverage these results in tools that efficiently combine all the different ideas. Above all it remains unknown how well the various approaches and partial solutions for dynamic symbolic and symbolic execution work together.

Contributions of this Thesis regarding Dynamic Symbolic Execution. In this thesis, I have worked extensively on pushing the boundaries of dynamic symbolic execution for Java programs. The work focused in particular on the dynamic symbolic execution engine JDart, when I was fortunate enough to be one of the team leading JDart to a gold medal at SV-COMP 2022 [21]. With the introduction of GDart in Paper V [101], we demonstrated that the concepts presented for JDart scale on an industry-grade JVM (The GraalVM) advancing the scalability of dynamic symbolic execution for Java. GDart bundles these items in more independent components. The main new component involving the GraalVM is the concolic executor SPouT presented in Paper VI [105].

This thesis extends JDart for dynamic symbolic execution of programs with strings, central for JDart's success at SV-COMP and its application in Jaint. The SV-COMP papers [102, 103] advertise the string extension to JDart on a high-level perspective. Section 4 describes these in greater detail and explains how JDart generates the string constraints. This represents a continuation of the work by Redelinghuys et al. [120], considering the advances in the field of SMT solving. Especially, the evaluation demonstrates how well today's string theory performs in SMT solvers for encoding string operations during dynamic symbolic execution. Paper VI [105] describes the string encoding in SPouT in more detail.

Moreover, this thesis extends JDart's SMT constraints representation and makes contributions in the area of using portfolio solvers in the search backend presented for SV-COMP 2021 [103]. Section 2.3 describes the impact of model selection in the SMT solver on dynamic symbolic execution and elicits additional quality requirements for the model selection of an SMT solver in the context of dynamic symbolic execution. Furthermore, it explains in futher detail the idea of bounded solvers for speeding up the performance of dynamic symbolic execution as presented briefly for SV-COMP 2020 in Paper III [102]. The other dimension we use for bounding the search space is the design of the driver method. Section 3.3 discusses the effects in greater detail and explains how this technique abstracts away the execution environment with symbolic models. Controlling which aspects of execution are deterministic and which are nondeterministic is central for using Jaint [107] for the analysis of Java web servlets.

1.1.3 SMT Constraint Solving

Analyzing web applications has a long history, and solving string constraints is especially important for analyzing web applications, as these often process data in string formats (e.g. [35, 41]). By their nature, web applications often use methods dealing with string data and operations on strings. Christensen et al. [41] were one of the first groups presenting reasoning on string operations in Java applications for security analysis. Automatons back the solving procedure. Later a proposal for a standardized SMT format [27] evolved into the current version of Unicode constraints in the SMT-Lib [15] language.

SMT solvers power many of the modern formal methods used in practice today. This has been possible as this field has advanced constantly over the last decade. Especially important for this thesis are the advances in the area of reasoning on string constraints (e.g. [9, 16, 17, 26, 56, 121]). As explained by Chen et al. [40] in their introduction to the Ostrich solver, there is, from a theoretical point of view, still a demand for increasing the reliability and efficiency of string constraint solvers. Many solvers apply heuristics with astonishing results but do not guarantee completion in every case. Chen et al. raise the question of whether the benchmarks used in the area of string solving are representative of the domain.

A follow-up question is how we should deal with SMT solvers in our tools to cover bugs in implementation (as found recently by SMT solver validators [28, 91, 128, 144]) and variation in performance due to the heuristics used (e.g. [117, 123, 145]). The answer is in general portfolio solving, but it is still an open question what kinds of portfolio solvers exist and when we should use them. Caching used to be considered a central point for improving SMT solving performance in symbolic execution. The Green framework introduced this idea [141] and Brennan et al. [33] conducted some follow-up work for caching string constraints. But caching requires constraint normalization and computing these normalizations is expensive. Given the often fast solving times of SMT solvers on the available benchmarks, caches have not become an answer to the problem of performance variation. Portfolio solving seems more promising.

Contributions of this Thesis regarding SMT Solving. This thesis does not contribute to the theory of SMT solvers or to the heuristics used. Instead, it focuses on building the required engineering knowledge for using SMT solvers in tools.

One insight is that the SMT language has become a fourth-generation language for tool builders applying formal methods. Consequently, in order to integrate them into our tools we need suitable abstractions that fit into the language used for programming the analysis. As part of this thesis, I contributed to the engineering of JConstraints, an abstraction library to work with SMT problems in Java. Paper VII [77] presents JConstraints for the first time as an independent library, and I contributed to the idea of comparing SMT languages with other fourth-generation languages embedded in programming languages. I contributed furthermore to the description of JConstraints in this paper.

Moreover, I worked on describing requirements for a constraint-solving service that would simplify the usage of SMT solvers in tools. Paper VIII [98] presents these ideas. This paper already mentions the need for empirical data to answer many decisions, especially to find the right fit between available resources, SMT solvers, and the workload profile of SMT problems. This question is especially important for running the dynamic symbolic execution of string programs. Paper IV [100] contains the contributions made in abstracting combination patterns for SMT solvers, mining the required data for making informed choices on the string solving capacity of a solver, and analyzing the data for engineering a good portfolio solver for dynamic symbolic execution of programs with strings. This includes a detailed discussion on the tradeoff between resource consumption and correctness. Section 2.4 will extend this discussion with concrete examples and tradeoff decisions leading to the portfolio solver for SV-COMP 2021 [103]. The experiments in Section 6.1.1 demonstrate the impact of different SMT solvers on dynamic symbolic execution.

1.2 Organization

The body of the thesis is organized into four chapters describing the contributions in the three areas in more detail, with a further chapter of evaluation and discussion before the concluding chapter. Chapter 2 presents the work, experiments, and insights developed in the area of applying SMT solvers for tool building. Chapter 3 describes the preliminaries for dynamic symbolic execution and implementation details of the architecture used for JDart and GDart. In continuation, Chapter 4 goes on to present the advances made for the dynamic symbolic execution of programs with strings and describes how the driver method used influences the soundness of the analysis presented. The Jaint framework is described in Chapter 5. This chapter first discusses in greater detail the taint analysis problem, and then the different components of Jaint and how they interact to realize the security analysis of Java web applications. The discussion of related work, where appropriate, is also embedded in these four chapters. Chapter 6 evaluates and discusses the contributions made in this thesis toward the research vision before Chapter 7 concludes the thesis and provides an outlook on future work.

2 SMT Solving

This thesis analyzes software programs by transforming the programs into logical abstractions and using automated reasoning to check whether properties encoded in the same logic hold on these abstractions. All these abstractions are expressed as Satisfiability modulo theories (SMT) problems, and SMT solvers decide these SMT problems. In Section 2.1, I will briefly describe an SMT problem from a tool builder's perspective, focusing on how SMT problems can be used in analyses. In Section 2.2 I will then briefly present the different theories defined in the standard SMT-Lib language required for analyzing JAVA programs.

SMT problems are a variation of the more traditional 3-SAT problem, known to be NP-complete. Therefore, decision procedures use heuristics rather than optimal deterministic algorithms, which rely on random decisions for exploration strategies to speed up problem-solving. From a user's perspective, this raises different problems: models might be chosen randomly, it is hard to predict a solver's resource consumption, and—independently of the theoretical outcome—solvers are allowed to give up on problems if the heuristic cannot solve them. Section 2.3 describes the bounded solving heuristic for influencing the model selection of solvers. This is a technique for improving interaction with an SMT solver as sketched out in Paper III [102]. Section 2.3 explains it in greater detail. Portfolio solving avoids the impact of the other two problems on tool architecture, as discussed further in Section 2.4. It briefly shows important aspects of building portfolio solvers discussed in detail in Paper IV [100].

2.1 Preliminary: SMT Problems

Programs use variables of different types. Each type T represents a different, potentially infinite set (called the domain D of the type) of allowed values that might be assigned to the variable during the execution. The program introduces restrictions on these variables during its execution. Because of the restrictions, it might be impossible to assign certain values of the original domain D_T to a variable in the future of the current execution path, e.g., if the program enforces an if-condition that an integer variable X is less than five. The SMT language allows one to model these restrictions and the impact of a program on the state space formed by its variables in logic. The resulting description in logic is an SMT problem.

SMT solving is a theorem-proving technique that allows an automated decision for an SMT problem P defined by a set of constraints C restricting the problem space (typically called the domain D) respecting the semantics defined in a set of theories T. I will define these terms below, but these are the main components of an SMT problem. A decision procedure solving such a problem is an SMT solver, and I will mention some of the main solvers and a survey on decision procedures at the end of this section.

SMT Problems. The constraints in C are defined in the form of functions (cf. Definition 2.1). The functions establish a relationship between members of the domain D. These members are either a concrete value from the domain—I call them *constants* in this thesis if the value is known or *variables* if the value is unknown—or another function. The SMT language uses extensively function composition to express more complex logical constraints, but every member in C must evaluate to a Boolean value. In the context of an SMT problem, the Boolean *true* value is often called *satisfiable* and the $false$ value *unsatisfiable*. With this small modification, the Boolean value and operations work within the SMT language as in any common Boolean algebra.

Definition 2.1. (Function) Functions are the basic building blocks for the constraint set C. They are always defined over a Domain D representing a mapping from a tuple of parameters to a single value. Consequently, the function signature looks similar to $(D \times \ldots \times D) \rightarrow D$, but the tuple might also be empty. D is allowed to consist of a set of subdomains, e.g., integers $\mathbb{Z}$ and the Boolean values $B = \{true, false\}$, but might be a single domain as well. The function signature might restrict the function to accept only certain subdomains of D as a type of parameter or the result value, e.g., $>: \mathbb{Z} \times \mathbb{Z} \rightarrow B$.

We differentiate two types of functions:

Constants and Variables Functions without parameters are called constants. They have the signature $() \rightarrow D$. There are two types of constants in the SMT-Lib language: Fixed concrete values (called *constants*) represented by a literal of the domain D and named placeholders for concrete values (called *variables*[1]).

Named Functions All functions with at least one parameter are called named functions. The domain tuple has the same size as the number of parameters.

All functions have in common that they are written in infix notation. The parameters follow the name. For example, the named function $f : (D \times D) \rightarrow D$ is used later as $(f\,0\,1)$, if the constants zero and one are passed as a parameter to f.

As described in Definition 2.1, functions without parameters are constants. The named placeholders work in our modeling as normal variables in any math problem. An SMT solver assigns values from the respective domain to these variables whenever it is possible to satisfy all constraints in the problem. As a consequence, it is often interesting which variables a given SMT problem P contains. $V(P)$ describes all variables in P.

Named functions can be either interpreted or uninterpreted. Uninterpreted functions are allowed on arbitrary domains, and the only consideration made by the constraint

[1] A single variable is represented as a single character in this thesis for better readability of constraints but arbitrary names are allowed as a variable's name in the SMT language.

solver is that the same parameters always generate the same output for one instance of a function. This property is called *functional consistency* (for more details on uninterpreted functions and functional consistency, I refer to Chapter 4 of Kroening & Strichman [84]). Therefore, SMT solving allows any elements in domain D at the core language, treating them as uninterpreted functions. Interpreted functions require an interpretation semantic. This semantic is sometimes called an *interpretation* in the general context of satisfiability (cf. Chapter 1 by Franco et. al in the Handbook of satisfiability [23]). For the purposes of the present thesis, I will call the interpretation semantic for one specific part of the domain a theory. As core component, SMT problems always contain the Boolean algebra together with the logical support function *distinct*, *equality*, *imply*, and *exclusive or*. Therefore, the Bool sort with the values *true* and *false* and an interpretation of the operations NOT, =>, AND, OR, XOR, =, and DISTINCT following the common logical interpretation is always available for any SMT problem. The SMT-Lib language [14], a standardized format for specifying SMT problems, calls these operations the CORE theory for SMT problems. Any other theory can be defined in addition to the CORE theory, but it is included in any SMT problem as default theory. Section 2.2 discusses some more available theories in the SMT-Lib language. The set of used theories defining the available interpretations for functions in an SMT problem is in this thesis called T.

Given that all the constraints, the domain and the theories for a problem are defined, the question to solve in an SMT problem is whether all functions in C can be evaluated truly at the same time on the domain D respecting the interpretation of named functions defined by the theories in T. Definition 2.2 summarizes the characteristics of an SMT problem.

Definition 2.2. (SMT Problem) An SMT problem P is a triple (C, D, T) where

- C is a set of constraints that should be satisfied,
- D is a domain defining the problem space of the problem, and
- T is a set of theories, for interpreting the constraints on the given theory.

The central challenge in an SMT problem is to answer whether all constraints in C are satisfiable. The answer is either yes (satisfiable or short *sat* and we write $(D, T) \models C$) or no (unsatisfiable or short *unsat* and we write $(D, T) \not\models C$). This answer always exists if the theory is decidable. If the theory is partially or completely undecidable, the answer for the specific problem might still be sat or unsat, but decision procedures are allowed to give up on the problem and return *unknown* as well.

Let's review these terms, then, on the following example, in order to understand a problem in more detail:

Example 2.1. For this example, I will use the theory of normal integer arithmetic without quantifiers. This theory is named QF_LIA in the SMT language. Domain D comprises all possible integers $\mathbb{Z}$ in conjunction with the Boolean domain part of the CORE theory in the SMT language. The theory allows the usage of functors that are linear integer operations: $+, -, \leq, <, >, \geq$, and in some corner cases multiplication $*$. The interpretation of these functors is congruent with the semantics of normal integer operations. Now, we can describe an example constraint set:

$$C = \{(<\ 5\ x), (\geq\ 19\ x), (=\ y\ (+\ x\ 4)), (\leq\ y\ z), (=\ z\ 13)\}$$

Combining the constraint set with the domain and the theory, we get a triple defining the problem $P_{exmp} = (C, \mathbb{Z}, \{QF\_LIA\})$. The problem describes a relation between three variables x, y, z bounding their possible values by using some constants $\{4, 5, 19, 13\} \in D$. The interpretations of the functions require that 5 should be less than x and x less or equal to 19. y should be less or equal to z and 4 greater than x. Furthermore, z should be equivalent to 13. The question is whether all these requirements can hold simultaneously, given that x, y, and z are integers.

Definition 2.2 speaks of the satisfiability of a constraint. Today's SMT solvers do not only answer whether a problem is satisfiable, but also compute a model that proves the satisfiability of the problem respecting the interpretation of different functors according to the theories and domains involved. Let us start by explaining the intuition behind the concept of a model using the problem P_{exmp} from the previous Example 2.1. Assume the following assignment M for the variables $V(P_{exmp})$: $M = (x \rightarrow 6, y \rightarrow 10, z \rightarrow 13)$. An SMT solver could claim that M is a model that satisfies P_{exmp}. To validate whether M is truly a model, every constraint $c_i \in C$ must evaluate to true using the assignment for the variables in M. $(<\ 5\ x)$ combined with the assignment $M(x) = 6$ evaluates to true, as 6 is greater than 5; $(\geq\ 19\ x)$ evaluates true, as 6 is less than 19; $(=\ y\ (+\ x\ 4))$ evaluates true, as 10 is equal to 6 plus 4; $(\leq\ y\ z)$ evaluates true as 10 is less than 13; and $(=\ z\ 13)$ evaluates true as 13 equals 13. As all constraints evaluate to true, M is a satisfying assignment for the problem in Example 2.1 and therefore a proof that the problem is satisfiable. Following the ideas presented by Nieuwenhuis et al. [115], Definition 2.3 defines a model for an SMT problem.

Definition 2.3. (SMT Model) An SMT problem P is satisfiable, if and only if there exists at least one assignment M defining for every variable in $V(P)$ a fixed return value from D, so that all constraints in C evaluate to true given the interpretations for the named functions in T. A satisfying assignment M is called a *model* for problem P written $M \models_{(D,T)} C$. If and only if no such assignment exists, an SMT problem P is unsatisfiable.

Normally, the variables in an SMT problem are existentially quantified and it is sufficient if one such assignment in the domain exists. Such problems are called quantifier-free problems. Quantifier-free theories are normally prefixed with *QF_* in their name. Sometimes it is necessary to formulate a constraint that mixes all quantifiers with existing quantifiers. Problems containing both sorts of quantifiers are harder to solve and therefore clearly separated in the theory definition. Quantified theories are usually not further prefixed.

SMT Solvers. Making the decision, whether a model exists or not for a given SMT problem is the task of an SMT decision procedure embedded in an SMT solver. In the 1960s, Davis and Putnam [55] presented a decision procedure for boolean satisfiability problems (later called SAT problems, in accordance with the common terminology in the literature). Davis, Logemann, and Loveland [54] improved the procedure shortly afterward and the resulting algorithm is known today as the Davis-Putnam-Logemann-Loveland (DPLL)algorithm. An SAT problem is equivalent to an SMT problem using only the boolean core theory. Roughly 45 years later, Nieuwenhuis et al.[115] described the extension of the DPLL algorithm to the DPLL(T) algorithm, the current state-of-the-art algorithm for architecting SMT solvers. The SMT solver community has made tremendous progress on building SMT solvers using this pattern in the last decades, most notably with the introduction of the widely used solvers Z3 [56] and cvc5 [13], the successor of CVC4 [16]. To homogenize the SMT solver front ends, the SMT-Lib language [14, 15] standardized the input language of an SMT problem together with the theories to be supported by the solvers. SMT-Lib is a tremendous step in making SMT solvers interchangeable with each other in tools and software projects relying on SMT solvers.

The abstraction of an SMT solver defined in the SMT-Lib interface allows treatment of SMT solvers as a black box in the context of this thesis. However, I will shortly introduce the idea of the DPLL(T) framework and refer the interested reader to the book by Kroening and Strichman [84] for a brief overview of examples of the underlying algorithms used in the theory solvers. The book provides only a brief overview, explaining the basics of some theories, e.g., uninterpreted functions, linear algebra, bit vectors, and arrays, but stopping short of the algorithms used in theory solvers for string constraints, an important theory in this thesis. This area is still under considerable development and has led to many available theory solvers in the last ten years that complement the explanation of the DPLL(T) framework by Kroening and Strichman, e.g., ABC [9], Z3str4 [96], Z3str3 [17], Trau [1], Norn [2], and the theory solver in CVC4 [121].

In the following paragraphs I will briefly discuss the basic idea of DPLL(T). The next section will discuss some of the included theories in SMT-Lib that are relevant for DSE as presented in this thesis, together with strategies for designing the theory solvers specific to these theories.

From DPLL to DPLL(T). DPLL takes as input an SAT problem and returns either a model M, which is an assignment for all variables in the SAT problem, or affirms that the problem is unsatisfiable if the algorithm terminates. Understanding DPLL requires prior discussion of two special types of constraints: unit variables and pure variables. Moreover, DPLL solves SAT problems in CNF encoding. I will explain these terms with an example:

$$(A \lor \neg B) \land (B \lor \neg C) \land (\neg A \lor D) \land E \tag{2.1}$$

Consider the SAT problem in Equation 2.1. As all clauses are connected by conjunction and each of the clauses only contains disjunction or a single variable, the problem is expressed in the conjunctive normal form (CNF). Thus, for example, the Tseitin transformation [140] allows the transformation of any SAT problem into CNF. There, the CNF limitation, as the expected input format to DPLL, does not limit applicability. The variable E is a *unit variable* as the clause is only satisfiable if E is assigned a true value. D and $\neg C$ only occur in these constraints as true values (e.g., D) or in the negated form (e.g., $\neg C$). Therefore, they are *pure variables*.

DPLL is a systematic search algorithm that tries out all possible assignments for the variables involved in the SAT problem. Nieuwenhuis et al. [115] describe the search heuristic using five transitions, and I will follow their idea in the following treatment, as DPLL(T) extends this transition system. Without defining them formally, the five transitions are UnitPropagate, PureLiteral, Decide, FailState, and Backtrack. DPLL starts with an empty model and assigns different variables step by step until the model satisfies the SAT problem or the search terminates without a satisfying assignment in the FailState transition. UnitPropagate satisfies a unit variable not yet assigned in the model and simulates the effect on the remaining clauses. PureLiteral satisfies a pure variable not yet assigned in the model and simulates the effect on the remaining clauses. Decide makes a decision and assigns any of the still unassigned variables to true together with an annotation that this assignment is the result of a decision. FailState terminates the search, if no further Backtrack is possible and the problem is not satisfiable within the current assignment. Backtrack reverts the model chronologically to the state before the last Decide transition. It then adds the false value for this variable to the model, as the true value cannot lead to a satisfying assignment. It is important to note that DPLL backtracks chronologically, and a bad decision made in the first choice might require the exploration of a large area of the problem space before it gets corrected. The priorities of transition are defined the same way that all UnitPropagate transitions must be enabled, before following the PureLiteral transtions. If neither, UnitPropagate nor PureLiteral is possible, a Decide or Backtrack transition will be used. If none of these is available, the FailState transition terminates the search. So let's solve Equation 2.1 using these transitions:

$$\begin{aligned}
\emptyset || (A \vee \neg B) \wedge (B \vee \neg C) \wedge (\neg A \vee D) \wedge E && \textsc{UnitPropagate}\\
E || (A \vee \neg B) \wedge (B \vee \neg C) \wedge (\neg A \vee D) && \textsc{PureLiteral}\\
E, \neg C || (A \vee \neg B) \wedge (\neg A \vee D) && \textsc{PureLiteral}\\
E, \neg C, \neg B || (\neg A \vee D) && \textsc{PureLiteral}\\
E, \neg C, \neg B, \neg A ||
\end{aligned}$$

In this example, the algorithm does no backtracking, because any decision is mandatory due either to a UnitPropagate or a PureLiteral transition. The resulting model $E, \neg C, \neg B, \neg A$ satisfies the original problem. The chronological backtracking is slow, and modern SAT solvers use a more advanced approach involving learning a conflict clause, so they can backtrack further than just the last chronological decision and add a clause containing the conflict leading to the backjump. This approach is known as the *conflict-driven clause learning* (CDCL) algorithm. Nieuwenhuis et al. [115] respect this in their transition approach by adding transitions for learning and forgetting clauses, as well as a more powerful backjump transition. They call the resulting transition rule set a *DPLL system with learning.* It describes the same approach as CDCL.

According to Nieuwenhuis et al. [115], DPLL(T) allows implementation in two flavors: the *eager* and the *lazy* approach. The eager approach translates the constraints into a boolean CNF problem using a satisfiability-preserving transformation. The resulting CNF formula can be checked by an off-the-shelf SAT solver. However, the transformation process is expensive and has to be completed before checking the resultant SAT problem. In practice, the lazy approach performs better, and modern SMT solvers follow this implementation strategy. The lazy approach replaces every theory related subclause with a Boolean variable and computes a satisfying model for the relaxation of the original problem. If the relaxation is unsatisfiable, the original problem is unsatisfiable as well. Otherwise, a theory solver is used for checking whether the conjunction of all theory subclauses is also satisfiable by a theory solver T as well (called the T-consistent check for a model). Otherwise, the theory solver has to explain the inconsistency with a theory lemma that excludes this invalid assignment in the future. With this lemma, the algorithm backjumps and either finds a new model to check or proves the problem unsatisfiable. As the lemmas are added to the original problem and guide the search in the boolean problem, the formalization for the lazy version of DPLL(T) is a variant of the DPLL system with learning. I will conclude this section, therefore, with an example of applying the lazy DPLL(T) algorithm to the constraints in Equation 2.2 involving linear integer arithmetic.

$$((>\ A\ 5) \vee (<\ B\ 3)) \wedge ((>\ B\ A) \vee (\neg\ (=\ C\ 4))) \wedge (>\ C\ 3) \tag{2.2}$$

$$\begin{gathered}(Z \vee Y) \wedge (X \vee V) \wedge W;\\ Z = (>\ A\ 5), Y = (<\ B\ 3), X = (>\ B\ A), V = (\neg\ (=\ C\ 4)), W = (>\ C\ 3)\end{gathered} \tag{2.3}$$

First, the lazy approach to DPLL(T) replaces the theory-related constraints with boolean variables as shown in Equation 2.3. Next, an SAT solver computes a model, e.g., using the DPLL algorithm. A possible model for this boolean problem is Z, X, W independent of the values for Y and V. Next, the integer theory solver validates whether the subproblem $((>\ A\ 5) \wedge (>\ B\ A) \wedge (>\ C\ 3))$ is satisfiable and the model is, therefore, T-consistent. This problem can be expressed, e.g., as a linear program solved with the simplex algorithm (cf., e.g., Linear Programming by Chvátal [42] for an introduction to linear programs and the simplex algorithm). As this subproblem is T-consistent, e.g., with the model M = {A = 6, B = 7, C = 4}, the equation is satisfiable and the DPLL(T) algorithm returns.

2.2 SMT-Lib Theories relevant for DSE

SMT-Lib defines seven major theories in version 2.6 [14] that the major SMT solvers provide ready to use for logical modeling: Arrays, FixedSizeBitVectors, Core, Floating-Point, Integers, Reals, and String. As described in the previous section, the core theory involves equality, uninterpreted functions, and Boolean operations. For analyzing JAVA programs as described in this thesis, we mainly require the FixedSizeBitVector, Floating-Point, and string theory. As string theory encodes numeric constraints using integers, Integer theory is technically also involved in the analysis but only for converting an integer to a fixed size bitvector; so I'll skip the presentation of integer theory here. The remainder of this section will describe the theories used and indicate the algorithms used in theory solvers.

FixedSizeBitvectors. Computers are discrete machines representing numbers in an encoding called Two's complement of a fixed size number of bits. Consequently, the integer theory does not represent arithmetic computation semantics in most programming languages (except when analyzing Python3 code). Instead, the fixed size bitvector theory represents these computations, including a representation of the underflow and overflow behavior observed in a CPU. Bit vectors of 32- and 64-bit size represent JAVA's integer and long values correspondingly.

The bit vector theory is well established and supported by many SMT solvers. The named functions interpreted in the context of the bitvector theory are all prefixed with *bv*, e.g., *bvadd* for adding two bitvectors or *bvand* for comparing bitvectors bitwise. Apart from Z3 and CVC4—both have established themselves in this area—many other bitvector theory solvers are available, e.g., Boolector [114] and BITWUZLA [112]. The baseline approach for solving bitvector constraints is flattening the bitvector operations to a boolean formula solvable by an SAT solver. This technique is colloquially also called *bit-blasting* (cf. Chapter 6.2 in Kroening and Strichman [84]). Bitvector theory specific solvers advance in this area by developing better algorithms for this theory(e.g., Niemetz and Preiner [113], Wintersteiger et al. [146]).

Floating-Point Theory. The analyses presented here use the floating-point theory to represent float and double values from JAVA in SMT problems. The floating-point expresses a value using some bits for a mantissa, some bits for an exponent, and an additional bit for the sign. This way, floating-point numbers are expressed using a bitvector. The JAVA language uses a 32-bit floating-point and a 64-bit floating-point value following the IEEE 754 encoding format. The SMT-Lib floating-point theory supports IEEE 754 encoding and a Float32 and Float64 data type that matches the semantics of the JAVA types. Named functions interpreted by floating-point theory are prefixed with *fp.*, e.g., *fp.add* for adding two floating-point values.

Brain et al. [31] give a good overview of different approaches for solving constraints in floating-point theory following the IEEE 754 format and presented SymFPU, the current floating-point solver of CVC5, today's successor of CVC4. They have shown that the bit-blasting algorithm in SymFPU is performance-wise the best approach for solving floating-point constraints at the moment. CVC5 led the quantified floating-point solving single query track of SMT-COMP 2021[2]. The SymFPU theory solver backs up the second best solver, which is CVC5's precursor CVC4. For quantifier free floating-point queries, BITWUZLA scored second in the single query track of SMT-COMP 2021[3].

Combination of Bitvector and Floating-Point Theory. Real-world applications often use floating-point and bitvector numbers side by side, requiring reasoning about SMT problems containing both theories. Sometimes, values are cast between these theories requiring reasoning on the same value in both theories. Most SMT solvers therefore support a variant or extension of the Nelson-Oppen procedure [110] for combining different theory solvers (cf. Chapter 10 in Kroening and Strichman [84] for an explanation of this procedure within their solver framework, and cf. Jovanović and Barrett [81] for an example of an extension paving the way for the current theory combination in CVC5.). Therefore, the JAVA analysis presented here therefore requires solvers that support both theories and their combination. SMT-Lib defines interpretation to the functions *fp.to_ubv* and *fp.to_sbv* for casting a floating-point value to an unsigned or signed bitvector. For the other direction, SMT-Lib interprets the function *to_fp* and *to_fp_unsigned.*

String Theory. Reasoning on string theory constraints is especially important for analyzing JAVA Web applications (cf., e.g., [35, 41, 125]) as, e.g., the HTTP protocol is string-based, and much information is exchanged as strings. String constraints can either restrict the string value in a content constraint or the length of the string in a numeric constraint. The string content is restricted using named functions interpreted in the string theory for string operations and the regular expression language. These function names start either with *str.*, e.g., *str.++* for the string concatenation, or *re.*, e.g., *re.++* for the concatenation of regular expressions. Further, there is the *str.in_re* function that checks whether a string content matches a regular expression combining the string and regular expression component of the string theory. The SMT-Lib maps the length constraint part on the numeric integer arithmetic, while the content part is mapped on the core string theory (cf. Bjørner et al. [27], Redelinghuys et al.[120]). In the more modern SMT-Lib version, the resulting theory is called *QF_SLIA* or sometimes *QF_SNIA* if nonlinear arithmetic operations are involved. Mixed quantifiers are not yet allowed in this theory.

JAVA uses int values to describe the string length modeled using 32-bit bitvectors. As SMT-Lib represents the string length using an integer value, comparing it with other int values in the logical encoding requires casting the value to a 32-bit bitvector first. Therefore, analyzing JAVA programs involving string requires tight linking between the FixedSizeBitVector theory and the integer theory in the respective SMT solver[4].

Both CVC4 [121] and Z3 [26] support string theory but there are also many theory-

[2] `https://smt-comp.github.io/2021/results/fparith-single-query`{last accessed: February 2022}

[3] `https://smt-comp.github.io/2021/results/qf-fparith-single-query` {last accessed: February 2022}

[4] *Side Note:* Interestingly, in the early proposals for the definition of SMT-Lib string theory [27], the numeric arithmetic is mapped to bitvector theory and early prototypes of string theory solver support the combination of string theory with bitvectors, e.g., Z3STRBV [137].

specific solvers around—e.g., ABC [9] and Z3str4 [96]—that can complement the capabilities of CVC4 and Z3 in the string solving area (cf. our study in Paper IV [100] on the performance of different string solvers). Regarding the algorithms, there are two main concurring approaches used in string theory solvers today: representing string constraints as automatons or as numeric sequences. To give two examples: the book by Bultan et al. [35] explains some ideas for solving string constraints using automatons as in ABC. In contrast, Bjørner et al. [26] explain the reduction of string constraints to integer problems implemented in Z3.

2.3 Bounded Solving

As defined in Definition 2.3, a model might be any appropriate assignment satisfying the constraints without any further quality requirements. This makes perfect sense in use cases requiring a yes-or-no decision but becomes a problem if the model provides the values for driving a concrete analysis. When I started to work with Falk Howar on extending JDart for the first SV-COMP 2020 submission leading to Paper III [102], we stumbled across this issue. One of the SMT solvers gave us large numbers in the SMT models leading to a large allocated array that exceeded the available memory. The solution sketched here adds additional bounding constraints to the original problems and modifies them with a heuristic. These bounds reduce the domain space, aiming for smaller values. The remainder of the section extends the argument to aspects of the model selection problem and its impact on dynamic symbolic execution. Further, it explains why bounding reduces the impact of this problem.

Example. Let us consider the example in Listing 2.1 for a better understanding of the effects. The listing defines a single variable X from bitvector theory with a width of 32 bits. Further, it contains a single constraint enforcing that this variable is greater than 8 encoded as a signed bitvector with 32 bits expressed in hexadecimal encoding. There are 2 147 483 639 possible values to make this assignment valid for a signed 32-bit bitvector given the bitvector theory. Moreover, each of these values is a valid solution to the problem. Using different SMT solvers, we get different answers as all solvers pick a single example out of this large domain. Z3 answers this problem with the model $M_{z3} = (X \rightarrow \#x00000008)$ which is the hexadecimal encoding for 8 and the smallest possible value in this constraint that might be assigned to X. CVC4 answers the model with the following assignment $M_{cvc4} = (x \rightarrow \#b01000000000000000000000000000000$, which is the binary encoding of 1 073 741 824 and somewhere in the middle of the possible value range. The bit pattern reveals a clear systematic bit flipping pattern from left to right. The highest-order bit must be 0 to maintain a positive sign. Otherwise, the constraint is violated. Setting the second highest bit to 1 solves the constraint. In terms of the SMT semantic, both models are successful solutions to the problem.

```
(declare-const X (_ BitVec 32))
(assert (bvsge X #x00000008))
```

Listing 2.1: Example of an SMT problem.

Next, it is important to compare the potential impact of these two different models on an analysis. Assume that this SMT problem in Listing 2.1 enforces that a given array should have at least eight entries and assume it is an int array costing four bytes per entry, the M_{z3} model creates a $8{*}4\,bytes = 32\,bytes$ array allocated while exploring the concrete path. Using the M_{cvc4} model costs $1\;073\;741\;824{*}4\,bytes = 4\;294\;967\;296\,bytes \,\widehat{=}\, 4\,GiB$. While the 32 bytes required for the small concrete array are negligible compared to the general memory requirements of a JVM and a modern SMT solver, the four GiB array consumes significant resources.[5] If the array allocates data structures more expensive per entry than an int in JAVA, the problem worsens until the array no longer fits into the concrete memory. As a side effect, the concrete run fails with an out-of-memory error making it impossible to analyze the given piece of software. The smaller model allows the concrete run to complete quickly and efficiently, collecting the new symbolic constraints that allow further symbolic exploration.

The Model Selection Problem. As shown in the previous example, models get a quality constraint that allows one to judge the suitability of a model for driving the concrete component of the analysis in comparison with another model. It depends on the concrete program under analysis whether it is more desirable to optimize the model for a smaller memory allocation or fewer loop unrolling steps during a potential concrete execution. Therefore, having a universally optimal model selection algorithm is impossible, and it is not easily possible to encode a constraint that optimizes the model. Nevertheless, from our overall experience, we have observed that choosing smaller values over larger ones tends to impact the analysis performance positively. Therefore, we use this as a selection heuristic and have introduced the bounded solving heuristic for generating models that use small values in the model of a satisfiable problem compared with all possible values in the domain.

Bounded Solving as Heuristic. Bounded solving is a heuristic that restricts the domain by limiting all variables to an interval range. We do this by expanding the problem with a statement, ensuring that each variable is less or equal to the positive value of the bound and greater or equal to the negative value of the bound. Only the variables from a numeric domain are bounded. Strings are not bounded regarding possible character values but might be bounded in length wherever suitable.

```
(declare-const X (_ BitVec 32))
(assert (bvsge X #x00000008))
(assert (bvsle X #x0000000a))
(assert (bvsge X #xffffffff))
```

Listing 2.2: Example from Listing 2.1 with bound 10.

Definition 2.4 (Bounded SMT Problem)**.** Given a SMT Problem $P = (C, D, T)$ and a bound b with $b > 0$, the bounded SMT Problem $P_{bounded_b} = (C_b, D, T)$ extends the constraints set C by bounding all $x \in V(P)$: x is in a numeric domain. The constraints set C is extended as follows: $C_b = C \cup \{(X \geq -b) \wedge (X \leq b) \mid \forall X \in V(P) :$

[5] *Side Note:* SV-COMP provides 14.6 GiB RAM to analyze a given task. So this single model has a tremendous impact on the analysis's overall performance as it already blocks 27 % of the available memory. Therefore, the model selection also influences the performance of our analyses in the SV-COMP challenge.

Algorithm 2.1 The Bounded Solving Heuristic

```
Input: SMT Problem $P$, List of Bounds $B$, SMT solver $S$
Output: SAT and model $M$, UNSAT and null, or UNKNOWN and null
procedure boundedSolving($P$, $B$, $S$)
    for b in B do
        $P_b = bound(P, b)$                ▷ Bound P with b as described in Definition 2.4
        $res, m = S.solve(P_b)$
        if res = 'SAT' then
            return res, m
        end if
    end for
    return S.solve(P)
end procedure
```

X is in a numeric domain$\}$. $\geq$ and $\leq$ are placeholders and have to be replaced to the equivalent in the theory associated with the domain of X (e.g., bvsge, bvsle, etc.).

Bounding the SMT Problem P alters the semantics of the original problem, and the results are only partially transferable. As we only add constraints in the bounding process limiting the solution space, any satisfiable solution for the bounded Problem P_b also satisfies P. Hence satisfiable decisions are transferable and the following holds for a model M that satisfies $P_b = (C_b, D, T)$: $M \models_{(} D, T)C_b \implies M \models_{(D,T)} C$. The proof is trivial as $C \subset C_b$. Therefore, if using the assignments for variables in M evaluates all functions in C_b to true, it also evaluates all functions in C to true. Unsatisfiable decisions are, in general, not transferable. If the P_b problem is unsatisfiable, this does not imply that P is unsatisfiable.

Wherever we use bounded solving during dynamic symbolic execution, we use a repetitive widening approach to solve the problem multiple times with increasing bounds, as shown in Algorithm 2.1. It takes a list of bounds B, an SMT problem P, and an SMT solver S and returns an answer to the problem P. The bounded problem is solved for each of the bounds in B. If it is satisfiable, the answer is returned. Otherwise, the next bound is tested. If the problem is not satisfiable for any of the bounds in B, the original unbounded problem P is solved. For unsatisfiable cases, the problem has to be solved $N + 1$ times in this approach ensuring the unsatisfiable result is valid. Satisfiable instances are solved 1 to $N + 1$ times. The heuristic favors satisfiable instances.

Related Work and Further Ideas. As I will demonstrate using the SV-COMP examples in Section 6.1.1, choosing the right bounds has a significant impact on the performance of the dynamic symbolic execution engine. For SV-COMP, we used empirical experiments to develop the bounds used in our tools today. Two theoretical ideas exist in related work that can improve bounded solving in future: UNSAT cores and optimization modulo theory (OMT) solvers. I will briefly present them below to complete the picture. However, as the solver component is not the bottleneck in the applicability of the proposed analysis after using the previously described bounding heuristic, I have

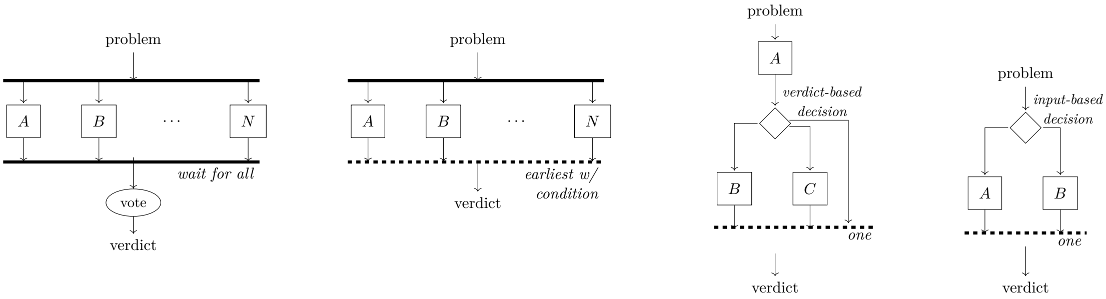


Figure 2.1: Four basic patterns for building SMT portfolio solvers as introduced in Figure 1 of Paper IV [100]. They are from left to right: majority vote, earliest verdict, verdict-based second attempt or validation, feature-based / capability-based solver selection.

not investigated their effects on solving times in any detail.

An UNSAT core UC is a subset of the constraints in an unsatisfiable problem $P = (C, D, T)$. The core contains contradictory constraints and is potentially smaller than the original problem (c.f. Chapter 2 by Kroening and Strichman[84] for a detailed explanation of UNSAT cores). If the UNSAT core UC_b of a bounded problem is a subset or equal to the constraints of the original problem, the unsatisfiable result of P_b implies that P is unsatisfiable as well: P_b is unsatisfiable $\wedge\, UC_b \subseteq C \implies P$ is unsatisfiable. This theoretical result can be combined with an SMT solver supporting UNSAT cores to reduce the number of solving attempts for unsatisfiable problems in the solving process, eliminating the discrimination of unsatisfiable instances in Algorithm 2.1. If the UNSAT core UC_b of the bounded Problem P_b is independent of the bounding constraints, solving the original problem without bounds is not required, as the original problem P is already unsatisfiable using the same core.

Optimization modulo theory (OMT) solvers (e.g., OptiMathSat [129] and νZ [25]) allow solution of an SMT problem optimizing the model for a specific target function. An OMT problem obligates the solver to find, for example, the smallest possible model for an SMT problem. As SMT decision procedures are not the main focus of this work, I have not investigated this direction any further nor run experiments with OptiMathSat or νZ.

2.4 Portfolio Solving

SMT solving is an area of constant development. Building a software verifier backed by SMT solvers requires dealing with this constant progress and uncertainty (c.f. our contribution to IEEE Software describing this challenge in more detail [78]). In Paper VII [77] and Paper VIII [98], we briefly discuss the means of decoupling the program from a single solver and how a solving service architecture fits into the general architecture instead of a single SMT solver. In this case, the SMT service layer idea is not coupled with the idea of a web service. It is just a meta-solving layer abstracting all the details of the individual SMT solvers and optimizing the selection process and problem-solving.

Paper IV [100] describes how to build such a meta-solver. Figure 2.1 shows the pattern proposed in this paper for combining SMT solvers in a meta-solving strategy.

The software verification community distinguishes tools supporting algorithm selection from those supporting portfolio techniques in the SV-COMP reports (e.g., The SV-COMP 2021 report [20]). The algorithm selection is attributed to Rice [122], where a procedure selects a single algorithm to solve the problem at the end. The portfolio technique runs a set of algorithms supposed to do the same thing and runs them to solve the problem. This is especially suitable if the verifier is not interested as to which concrete SMT solver answers the problem. We find these two approaches in two of the four patterns. The solver selection pattern on the right of Figure 2.1 represents the idea of algorithm selection, while the earliest verdict pattern, second from left, represents the idea of portfolio solving. In my opinion, the verdict-based second attempt pattern combines these two theoretical approaches. It is a mixture of algorithm selection and sequential portfolio solving, as the different solvers run in sequential order and the decision on how to proceed depends on intermediate results. Majority voting, the pattern on the left of Figure 2.1, is inspired by the N-Version programming approach introduced by Avizienis and Chen [8]. This pattern focuses mainly on increasing the accuracy of the meta-strategy, and it can detect errors in automated decisions early in the process and in a fully automated way.

Portfolio solving is especially important for analyzing programs with string operations as the support for the string theory is under intense development at the moment in different SMT solvers. Paper IV [100] shows the performance of different string theory solvers on a cleaned benchmark set in this area. The data shows that the proposed verdict-based second attempt strategy using CVC4 and Z3 outperforms each of the solvers involved on the benchmark set in terms of total solved problems, and is only slightly more expensive than CVC4 on its own. Since 2021 we have, therefore, used this meta-solving strategy in SV-COMP as the backend of our dynamic symbolic execution tools. The implementation of the strategy is publicly available as part of JConstraints[6]. It is an efficient use of the available compute time in SV-COMP. Otherwise—from the insight gained in the experiments, and given the performance data—I highly recommend, for analyzing unknown SMT workloads, use of one of the other strategies involving more solvers that are also more expensive in terms of computing time.

For the operational usage of software verifiers, Paper IV [100] demonstrates that it is crucial to benchmark the involved SMT solvers in a repetitive interval estimating the potential error margins of the verifier verdicts. Depending on the result of the analysis and the use case, it is worth investing more time in solving and validating the solution of constraints problems, or sacrificing the increased guarantees created by cross-validation in favor of performance.

[6] `https://github.com/tudo-aqua/jconstraints` {last accessed: February 2022}

3 Dynamic Symbolic Execution

Symbolic execution is a well-known software analysis technique for software testing proposed by King [82] for the first time more than four decades ago. The central point is expressing the program execution symbolically, getting a precise symbolic description of all possible execution paths. Combining this symbolic description of the program with a property encoded in the same symbolic logic allows checking of the property by solving the resulting decision problem. Today, these decision procedures are often SMT solvers, and the problem is encoded using SMT-Lib language. The analyses presented in this thesis use SMT-Lib language, but other state-of-the-art tools, e.g., CBMC [43], use a symbolic representation that is expressed in a bit precise SAT problem solved by a SAT solver at the end. Here CBMC follows the 1990s trend of using bit precise representations that can be solved either by binary decision diagrams (BDDs) (e.g., Burch et al. [36]) or an SAT solver (e.g., Biere et al. [24]).

I will not present all the details of symbolic execution in this thesis, as it is a well-established technique today, and surveys such as Baldoni et al. [10], or Cadar et al. [38] explains it very well. Instead, I will briefly compare dynamic symbolic execution with full symbolic encoding and symbolic execution, as (depending on context) all of them are sometimes called symbolic execution in the literature. Next, I will describe the general layout of the dynamic symbolic execution engine as we use it for GDart and JDart. This section focuses especially on required functions in the JVM for implementing a dynamic symbolic execution engine. The final section in this chapter, describing the tradeoff involved by using a concrete driver for dynamic symbolic execution analysis, demonstrates why this is a powerful tool for reducing search space during symbolic exploration but has, at the same time, the potential to weaken the analysis result.

3.1 Full Symbolic Encoding vs. Symbolic Execution vs. Dynamic Symbolic Execution

The popularity of symbolic execution is rooted in the symbolic encoding of the complete program in a single problem. After encoding the problem, it is possible to combine the symbolic description of the program with different properties allowing multiple analysis targets to be checked without reencoding the problem. This leads to a strong reasoning power if the symbolic decision engine is powerful enough to solve the resulting logical problem. A further strength of symbolic representation is the description of many states by symbolic constraints (c.f. [36]). The weakness is that not every problem can be described in a finite symbolic representation for all possible inputs, e.g., looping over a symbolic data structure cannot be described in a finite set of symbolic constraints.

This is called the path explosion problem (cf. [38]) and is often voiced as the main drawback of this technique. Bounding the state space to reason on a limited finite state space is one common way of dealing with this problem in industrial (e.g., [47]) and academically settings [43]. As long as a concrete program stays within the bounds, the verification result holds during operations. This converts a part of the verification effort into a bounds-checking problem as the bounded subset of the state space is usually smaller than the unbounded counterpart. The bounds-checking problem proves that the assumed bounds are not violated during execution of the program. This check could be run by analysis upfront or as a monitor during runtime.

The main difference between the symbolic encoding of a bounded model checker as CMBC, ideal symbolic execution as presented by King [82], and dynamic symbolic execution is the proportion of the program that is encoded symbolically when property checks are performed. The literature sometimes uses the term symbolic execution indiscriminately for symbolic encoding generated either by the bounded model checker CMBC or by King's symbolic execution. But, as I will show next, these are significantly different processes. In this thesis I will, therefore, use the term *full symbolic encoding* to describe the encoding technique used by CMBC and the term *symbolic execution* for the encoding generated by the techniques described by King. This section will explain the differences in more detail.

Bounded model checkers like CMBC [43] often generate a full symbolic encoding as a large disjunction of all program paths before checking whether properties hold on a specific path. Impossible path constraints do not influence the logical verdict of the overall disjunction between all paths, and the reasoning ignores them. One invocation of the decision procedure filters executable branches and checks properties in one large logical problem. In contrast, the ideal symbolic execution described by King [82] encodes only feasible paths in a program and checks whether both branches are possible at any branching statement. If this is the case, the branch statement creates a disjunction between the two paths in the encoding. Otherwise, the symbolic executor follows only the feasible path. Creating the encoding of the program does not require executing it. King has not specified when properties are checked and how the symbolic tree should be used.

In contrast to Figure 3.1, describing the constraint tree resulting from symbolic execution, dynamic symbolic execution does not require the full symbolic description of the program to be generated in order to check the property. Instead, it only generates the symbolic description for a single path that is concretely executed and checks properties along this path before executing the next path. Reasoning happens after executing each path to determine the input values for driving the next path using the collected path constraint. For dynamic symbolic execution, the logical checks are only applied for encoding effects along a single path. I will demonstrate this difference by example, first for a full symbolic encoding, followed by King's symbolic execution, and by dynamic symbolic execution. The demonstration focuses on describing the main differences between these three encodings, as all of them are referred to as symbolic encoding in their specific domain but have small differences. As all have in common that they need symbolic encoding for expressing the program symbolically, I will briefly explain the symbolic

```
public boolean foo(int a){
  a = a + 6;
  if(a > 100){
    if (a < 50){
      assert false;
    }
    return true;
  } else if(a == 100){
    assert false;
  } else{
    return false;
  }
  return false;
}
```

Listing 3.1: Running example method FOO to demonstrate symbolic execution.

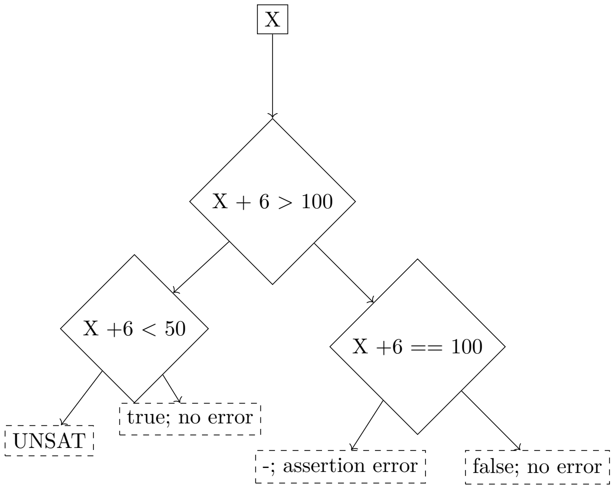


Figure 3.1: The symbolic execution tree of method foo. The symbolic variable X represents parameter a.

```
public boolean foo(int);                            1
  descriptor: (I)Z                                  2
  flags: (0x0001) ACC_PUBLIC                        3
  Code:                                             4
  stack=2, locals=2, args_size=2                    5
     0: iload_1                                     6
     1: bipush 6                                    7
     3: iadd                                        8
     4: istore_1                                    9
     5: iload_1                                     10
     6: bipush 100                                  11
     8: if_icmple 33                                12
    11: iload_1                                     13
    12: bipush 50                                   14
    14: if_icmpge 31                                15
    17: getstatic      #8                           16
    20: ifne       31                               17
    23: new            #9                           18
    26: dup                                         19
    27: invokespecial  #10                          20
    30: athrow                                      21
    31: iconst_1                                    22
    32: ireturn                                     23
    33: iload_1                                     24
    34: bipush 100                                  25
    36: if_icmpne 53                                26
    39: getstatic      #8                           27
    42: ifne       55                               28
    45: new            #9                           29
    48: dup                                         30
    49: invokespecial  #10                          31
    52: athrow                                      32
    53: iconst_0                                    33
    54: ireturn                                     34
    55: iconst_0                                    35
    56: ireturn                                     36
```

Listing 3.2: Running example method FOO to demonstrate symbolic execution.

encoding of JAVA programs upfront.

Symbolic Encoding of JVM bytecodes. Let me demonstrate the symbolic encoding of JVM bytecode by example for the simplified method FOO in Listing 3.1. The method takes a parameter a, adds 6, and compares it in various nested *if* statements. Some of the branches contain an *assert false* statement that terminates the program's execution. The other branches return either true or false, and the program continues with the return value without other side effects.

Listing 3.2 shows a human-readable representation of the JVM bytecode representing method FOO after compilation. Line 1 shows the method name, Line 2 to Line 5 represent metadata required for method look-up and memory reservation on the stack during the execution of method FOO. Line 6 to Line 9 are the bytecodes representing the addition of a with six and the reassignment to a in the first line of foo. We see that *iload_1* loads the parameter a on the method stack, *bipush 6* pushes the constant 6 on the method stack, and *iadd* computes the actual addition on the stack. The sequences of *iload_1*, *bipush N*, and *if_icmpM L* are the condition checks. N depends on the value pushed on the stack. The constant is taken directly from the condition checks in Listing 3.1. M is either *le* for less or equal, *ge* for greater or equal, or *ne* for not equal. L describes the jump target in terms of bytes, if the check is successful, e.g., the *if_icmple 33* bytecode in Line 12 continues with the *iload_1* bytecode in Line 24 if a is less than or equal to 100. Otherwise the next bytecode in line is executed. The sequences *iconst_0* or *iconst_1* followed by *ireturn* are the bytecode counterparts for the Boolean return statements. The two remaining sequences in Line 16 to Line 21 and Line 27 to Line 32 are the bytecode expansions of the assertions in Line 5 and Line 9 of Listing 3.1.

Let us assume that it is possible to annotate any value within the JVM with additional symbolic annotations ignoring the concrete realization. For encoding the bytecode sequence symbolically, we need to define the symbolic representation for three kinds of bytecodes: those that introduce atomic symbolic values (new variables and constants), those that create a functional relation between existing functions, or atomic values, and those that branch the control flow. In the context of this thesis, all of these are mapped on different types of functions that are used for defining the constraints of an SMT problem as described in Section 2.1.

The first category, bytecodes that introduce atomic symbolic values, is represented either as variables (aka functions without parameters) in the constraint of the corresponding SMT problem, or as constant values. For the example, we will symbolically represent the parameter a by the value X. The function type or constant type is chosen depending on the type of the corresponding constant or variable in the JAVA program: e.g., primitive int variables are represented as 32 bit bitvectors; therefore, X is a 32 bit bitvector. The JVM variables are annotated with the symbolic counterpart in SMT-Lib language.

The second category, bytecodes that establish a functional relation between other elements, is mapped on named functions from the corresponding theory. In the given example, the *iadd* bytecode in Line 8 of Listing 3.2 creates such a relation and is mapped on the semantically corresponding named function *bvadd* in FixedSizeBitvectors theory. As a is annotated with the SMT-Lib variable X and 6 is a constant, the annotation added to the result value of the addition is $(bvadd\, X\, 6)$[1]. Directly after the execution of the *iadd* bytecode in Line 8, the annotated sum is only placed on the stack, but the next bytecode in Line 9, *istore_1*, ensures that the reassignment to parameter a happens.

The third category, bytecodes that branch the control flow, is also mapped to named functions for the branching condition itself: e.g., the *if_icmple* bytecode in Line 12 of Listing 3.2 is mapped on the named function *bvsle*. The bitvector theory interpretation compares the two parameters against each other, interpreting the parameter values as signed bitvector numbers returning a Boolean value. This part of the semantic precisely

[1] Of course, 6 has to be encoded as 32 bit bitvector as well, but for readability, I use the decimal value here.

maps the JVM semantic of *if_icmple*. The condition might either hold or not during the execution of the path. The concrete semantic of the JVM never allows both events to be dealt with simultaneously in a single concrete run. How this split in execution is encoded is the main difference between the three symbolic execution techniques, and I will now discuss this in further detail.

Full Symbolic Encoding. The full symbolic encoding, e.g., the one computed by the symbolic executor inside the bounded model checker CMBC [43], unrolls and encodes all paths in the program until either all paths have been explored or an unwinding bound is hit. Whenever the encoder encounters a branching statement, the current symbolic encoding is forked and both branches are symbolically encoded. Encoding the effect of method FOO in Listing 3.1 as *res* and the information whether an error is raised as *error*, the full symbolic encoding for the problem appears as follows:

$$\begin{aligned} ((X+6) > 100) \wedge ((X+6) < 50) \wedge \mathit{error} \\ \vee((X+6) > 100) \wedge \neg((X+6) < 50) \wedge \mathit{res} \wedge \neg\mathit{error} \\ \vee\neg(X > 100) \wedge (X == 100) \wedge \mathit{error} \\ \vee\neg(X > 100) \wedge \neg(X == 100) \wedge \neg\mathit{res} \wedge \neg\mathit{error} \end{aligned} \tag{3.1}$$

The first path in Equation 3.1 with the path constraint part $((X+6) > 100) \wedge ((X+6) < 50)$ is not satisfiable, but as the different paths in the full symbolic encoding are all disjoint, this has no effect on the overall verdict in a property check. Maintaining the unsatisfiable paths also allows the entire symbolic encoding to be computed by following the bytecode without needing to involve an SMT solver. Loops are unwound until bounds are hit, eventually adding a large volume of unsatisfiable subconstraints to the overall disjunction that symbolically describes the program's effects.

Symbolic Execution. In contrast to full symbolic encoding, symbolic execution checks on every branching statement which of the branching conditions is satisfiable with respect to the current collected symbolic manipulations on the variables that are involved. 'Symbolic manipulations' refers to calculations that change the original symbolic variable before reaching the condition check. In the example in Listing 3.1, this refers to adding 6 to the original parameter a and eventually applying restrictions in previous path constraints. For example, the *if* condition check in Line 3 of the listing has only the symbolic manipulation on a that the value has been incremented by 6. Therefore, the current execution context is forked and one branch with $((X + 6 > 100))$ and one branch with $\neg((X+6 > 100))$. Inside the body of the *if* statement, the symbolic restriction $((X+6 > 100))$ applies for the parameter a. In consequence, $a < 50$ is unreachable, and the symbolic executor executes only the *else* branch of this condition check. The symbolic execution engine uses an SMT solver while exploring the program to make these decisions.

For symbolic execution, it is common to describe the encoding as a constraint tree. Figure 3.1 describes the constraint tree for method FOO in Listing 3.1. We clearly see the unsatisfiable path and all forking points during symbolic execution. King states that the symbolic encoding for the overall program is the disjunction of all reachable paths. The symbolic execution encoding of the same method therefore looks as follows:

$$\begin{aligned} &((X+6) > 100) \wedge res \wedge \neg error \\ \vee\neg&((X+6) > 100) \wedge ((X+6) == 100) \wedge error \\ \vee\neg&((X+6) > 100) \wedge \neg((X+6) == 100) \wedge \neg res \wedge \neg error \end{aligned} \tag{3.2}$$

In a direct comparison between Equation 3.1 and Equation 3.2, we see that the encoding is reduced by the unreachable path as the symbolic execution is a filtered view of the full symbolic encoding. Nevertheless, the full execution is symbolically encoded, and all intermediate calculations are visible in the encoding. The result is also symbolically encoded, allowing one to check properties for the whole program on the symbolic encoding in Equation 3.2. SPF implements this kind of symbolic execution for Java [118].

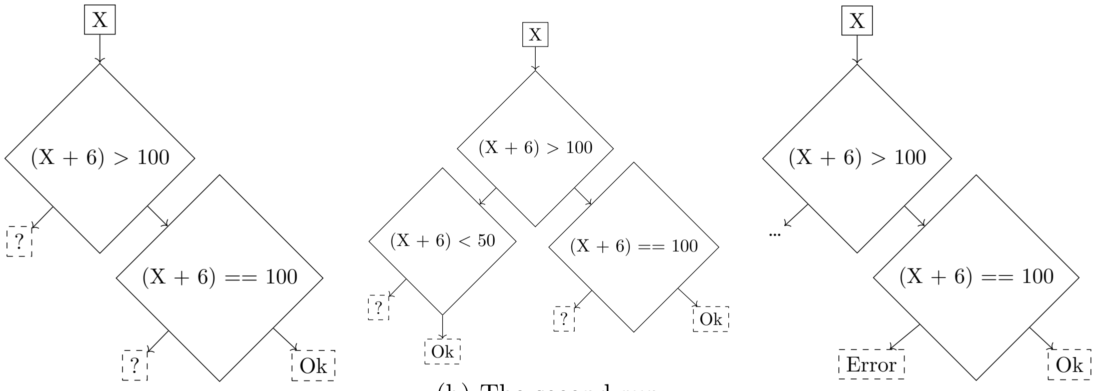
Dynamic Symbolic Execution. Dynamic symbolic execution also cleans up the unreachable branches in the symbolic encoding. While most dynamic symbolic execution engines allow collection of a symbolic constraint tree of the overall program and have some recording capabilities for computing a symbolic description of the complete program from the constraint tree, the analysis itself does not require the maintenance of the complete tree. Moreover, in contrast with symbolic execution, it uses the concrete run to decide at every branching statement the direction of the current execution branch. It obtains the symbolic program description from running the program concretely while recording all operations symbolically on the variable of interest occurring in the byte code. Decisions on further required concrete runs are computed by a symbolic decision-making component that works on the symbolic constraint obtained. Before I explain how this decision-making component works, we have to look at the symbolic constraint obtained from the first run, demonstrating the data underlying the decision-making.

Concretely running the method under analysis requires a driver that allows control of the input for symbolic variables. In the case of the *foo* method in Listing 3.1, a main method invoking *foo* might serve as a simple example of a driver omitting the details of boilerplate code for the value injection here. A first run is typically executed with some predefined default value for integers. The dynamic symbolic execution engines used for the thesis start to run the program with every numeric value set to zero and strings set to the empty string in the first run. For the *foo* method, we would observe successful execution of the concrete path and obtain a symbolic trace similar to the following:

$$\neg((X+6) > 100) \wedge \neg((X+6) == 100) \tag{3.3}$$

The main difference between Equation 3.3 and the previously presented encodings is the encoding of the single path. All other symbolic encodings of internal states are dropped after executing the path. The outcome is not symbolically encoded as only a successful execution or observed errors are recorded in the summary.

Next, the path constraint in Equation 3.3 is a conjunction of conditions along a single path. Each subpart of the conjunction describes one condition along the path that



(a) The first run.

(b) The second run.

(c) The third run.

Figure 3.2: The symbolic execution tree as unrolled during dynamic symbolic execution in the symbolic decision maker running breadth-first search. Each sub-figure shows the information added by a run.

potentially guards the execution of a second branch after the branching instruction. The symbolic decision component splits these individual conditions but maintains the order from left to right. By negating the individual conditions and solving the resulting path constraint using an SMT solver, the symbolic decision-maker computes concrete values for driving down the execution along the other paths in the program.

Figure 3.2a shows the internal representation of the trace from Equation 3.3 and the two branching points. The representation contains only the conditions and the information that the path executed successfully without errors. We see two not yet explored branches marked by a question mark in the representation. These are potentially executable branches representing new paths through the program. They are sometimes also called open branches in the temporary decision tree of the dynamic symbolic execution. The symbolic decision-maker picks one of these open branches according to its exploration strategy and tries to satisfy the condition. I will explain the different exploration strategies in the next section but assume for now that it takes the top branch for further exploration. After solving the SMT problem consisting of the single constraint $(X + 6) > 100$, the model returned by the SMT solver is transformed into a concrete value for reexecuting the program and driving the execution down this path. A concrete value for the parameter a satisfying this constraint is 100. Equation 3.4 summarizes the information obtained symbolically during the second run with $a = 100$:

$$((X + 6) > 100) \wedge \neg((X + 6) < 50) \tag{3.4}$$

The first condition in the trace maps to an existing decision node, this time taking the previously open branch as expected after solving the SMT problem. The path evaluates successfully without any observed errors or property violations and adds a new decision node to the tree. Figure 3.2b represents the updated constraint tree. The newly added node has a newly open branch. Assuming the symbolic-decision maker targets this branch next, it will create the SMT problem with the constraint $((X+6) > 100) \wedge ((X+6) < 50)$, but the SMT solver determines this problem as unsatisfiable. This implies that this branch is unreachable in the program. As this concludes all possible branches on the left of the root decision, this part of the tree is completely explored without any property violation or reachable error. In theory, this part of the tree can be replaced with a short remark that the state space behind this branch has been explored successfully. Figure 3.2c therefore only show three dots as replacement. In practice, the dynamic symbolic execution tools for Java used in this thesis maintain the complete constraint tree representing all explored branches. However, by the design of the analysis, they are not required to do so. The trees are maintained solely for documentation purposes.

Dynamic symbolic execution has to rerun the program a third time with 94 as concrete value for a to explore the last missing branch. This time, the symbolic trace will record the two already known decision nodes followed by an assertion error during execution. The path is marked accordingly, and the exploration stops, as the constraint tree is completely explored (cf. Figure 3.2c).

Comparison of Figure 3.2 and Figure 3.1 shows that dynamic symbolic execution represents potentially less information in the tree compared to a complete symbolic execution tree of a single method, as this technique does not require maintaining the complete tree in memory. Therefore, it is valid to materialize only a partial constraint tree over time[2]. While this looks at first glance like a weakness of the method, it is in fact the strength of dynamic symbolic execution over symbolic execution. Thus for analyzing the question whether *foo* can trigger an assertion violation, tracking the result of *foo* in the symbolic encoding is irrelevant. Of course, if contained in the symbolic execution tree, it does not prevent analysis of this problem. However, the additional symbolic encoding might add complexity to the decision problem, thus hindering the analysis. This is especially the case if full symbolic encoding is used for property verification, as larger aspects of the encoding are passed to the SMT solver. Moreover, as dynamic symbolic execution checks this property on each path, it uses symbolic reasoning power to compute the driving inputs but evaluates the property check dynamically in the JVM. If symbolically encoded properties are checked on the state space, they are checked after every path, as this is the only point during dynamic symbolic execution when the full symbolic encoding of the current path is available in the JVM. Symbolic execution and techniques working with full symbolic encoding must use symbolic reasoning power for larger encoded chunks of the program.

As I see it, dynamic symbolic execution is a divide-and-conquer approach for symbolic analysis. It not only slices the problem into smaller symbolic subtasks but can offload parts of the analysis into the runtime, e.g., the decision about which branch to follow first

[2] *Remark:* GDart and JDart maintain the whole explored part of the constraint tree because we have not yet encountered a situation where reducing tree size is required. Moreover, JDart encodes sufficient information in the constraint tree for computing symbolic method summaries from the constraint tree for historical reasons [59]. These features are extensions to a pure dynamic symbolic execution engine.

at a branching statement. Symbolic execution invokes the SMT solver at any branching statement to solve this question. This also introduces drawbacks, as analysis parts are pushed into the JVM and do not persist as an intermediate format by design. Collecting intermediate results for proof validation might require explicit infrastructure, whereas other techniques reuse intermediate results. By design, and in consequence of the divide-and-conquer strategy, dynamic symbolic execution has a chance to detect errors in the analysis of a single subtask along a single path. Therefore, it can already provide benefits in situations that do not terminate during the conversion to full symbolic encoding. Morover, the analysis can be used as a verification result if the search is exhaustive and all generated subtasks are solved. Without any changes in approach, it might be used in other cases as a white-box fuzzer for finding errors, with the proviso that guarantees of the absence of error are weaker if the technique is used for fuzzing (one example of successful application in fuzzing is the tool SAGE [70]). JDart and GDart are dynamic symbolic execution engines for the JVM, and I will go on now to describe common aspects of their architecture.

3.2 A Reference Design for Dynamic Symbolic Execution

As part of the present thesis, I participated both in the enhancement of JDart (cf. Paper III [102] and Paper II [103] in comparison to the original JDart paper by Luckow et al. [89]), and in the introduction of GDart (cf. Paper V [101]). Both are dynamic symbolic execution engines built on top of two different JVMs. They follow similar design principles in their architecture, but these are only partially discussed in the existing literature and are not presented in any of the papers mentioned above. JDart was mentioned for the first time in 2015 [59]], but the first detailed tool paper was released at TACAS 2016 [89]. Working on JDart led to participation at SV-COMP 2020 and the Jaint framework. In parallel, while working on JDart, an understanding of reference architecture for a dynamic symbolic execution engine arose between the participating developers. In this section, I will sketch that architecture and the mechanisms accumulating ideas behind both dynamic symbolic execution engines. Some details are still implemented differently, but the overall design is similar. I will explain this in more detail in the tool architecture section below. This will be followed by an explanation of some details of the symbolic explorer and the concolic executor. The remaining paragraphs in this section will describe specific technical details in the design of GDart and JDart that work similarly between the two JVMs and ease symbolic annotation during concrete execution. In addition, I will highlight design decisions in the Java standard library that are beneficial for the symbolic instrumentation. These can be safely ignored by readers not interested in implementing a dynamic symbolic execution engine for the JVM themselves.

Tool Architecture. Figure 3.3 describes the architecture of JDart. The main components are the *Explorer* and the *Executor*. We also find both components in the form of the DSE component (*Explorer*) and SPouT (*Executor*) in Figure 3.4, which describes the GDart architecture. In both tools, the explorer controls the execution of a certain

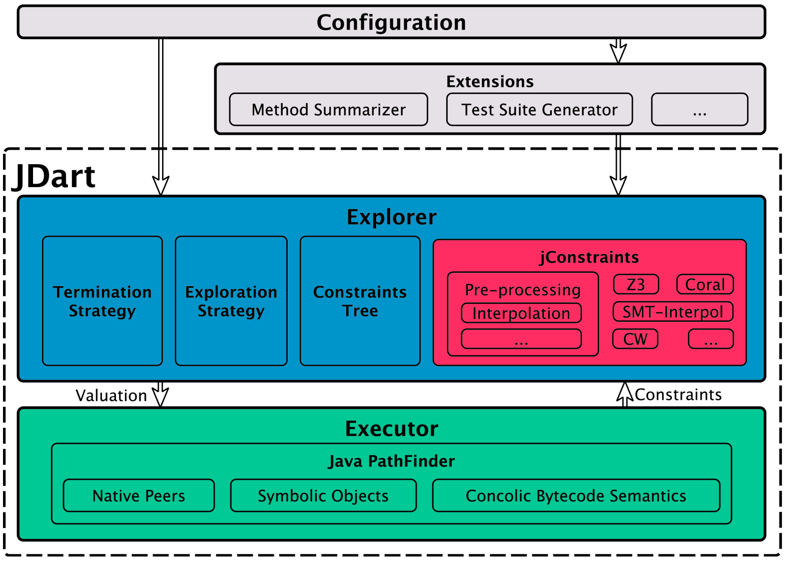


Figure 3.3: The architecture of JDart as described by Luckow et al. [89]

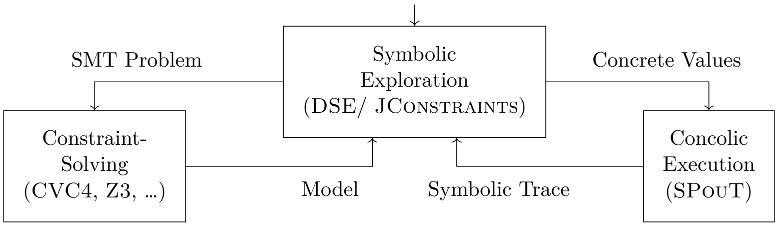

SMT Problem
Symbolic Exploration (DSE/ JConstraints)
Concrete Values
Constraint-Solving (CVC4, Z3, ...)
Model
Symbolic Trace
Concolic Execution (SPouT)

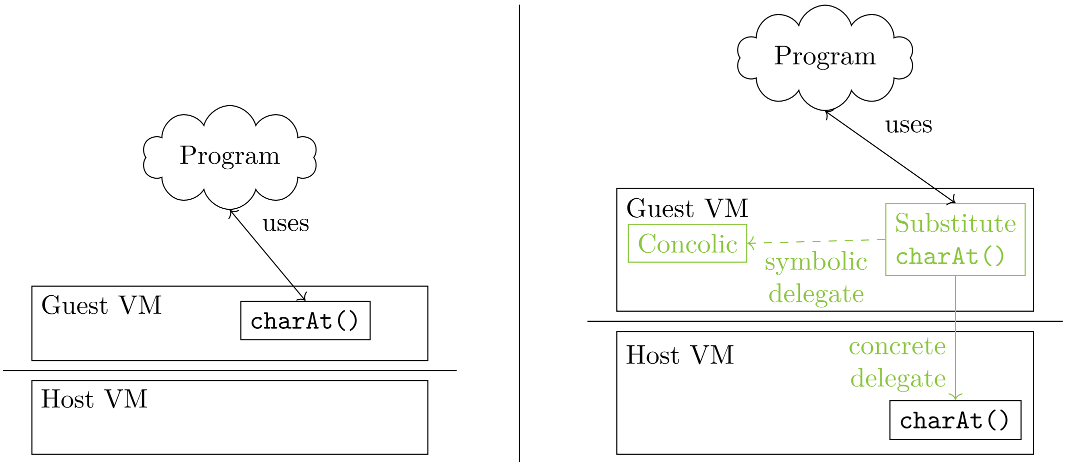
Figure 3.4: The architecture of GDart as described in Paper V [101]

.

branch by passing concrete values for concolic variables to the executor. The concolic executor reports back the symbolic constraints describing the execution path. Both tools use the JConstraints [77] library to represent the symbolic constraints in the constraint tree maintained by the explorer. Backend solving is done by a constraint solver, e.g., a state-of-the-art SMT solver like Z3, or a portfolio solver using the design patterns presented in Paper IV [100]. Both architecture sketches describe how the different components interact and which data is interchanged. The only difference between the two tools in this view is that JDart communicates by method calls as both components run within the same virtual machine, while GDart communicates between the components using inter-process communication and a string-based representation format for exchanging the gathered symbolic constraints[3]. A side effect is that JDart expands the constraint tree step-wise whenever a branching condition is encountered, while GDart

[3] *Remark:* For conciseness, I have not presented the string-based format in this thesis as it is a minor technical detail for the problem described. Nevertheless, the string-based representation documents the intermediate results produced by the executor and enables persiting them. Paper VI [105] describes the BNF grammar for the string based representation generated by SPouT.

updates the tree after every path. But this is a minor detail not influencing the fact that both tools start processing on the tree after finishing a path run. JDART does not exploit the earlier notifications on changes in the constraint tree compared with GDART's updates at the end of the path. Thus—if you ignore the names of concrete tools in the brackets—Figure 3.4 describes the reference design for a dynamic symbolic executor. The core components of JDART responsible for dynamic symbolic execution can be projected from 3.3 onto Figure 3.4.



(a) Normal proccessing of programs running on the guest VM.

(b) Using the implementation in the host VM for computing concrete effects.

Figure 3.5: Methods are replaced in the guest VM for symbolic instrumentation, but the concrete implementation in the host VM is reused.

Symbolic Explorer. TThe symbolic explorer maintains the intermediate constraint tree described for dynamic symbolic execution in the previous section. The main point to explain here is the different tree exploration strategies. The main algorithms used in this thesis for deciding the next branch are either backtracking or breadth-first search. Backtracking starts at the end of the last explored path and crawls upwards on the tree until a decision node with an open end is found, and the condition leading to this open end is satisfiable. Breadth-first search explores the tree from the root node, searching for the closest reachable open end in relation to the root. It is possible for both strategies to close multiple unreachable open ends before finding the next reachable path in the constraint tree. JDART originally supported only the backtracking implementation. Paper II [103] adds the breadth-first search as a configuration option for SV-COMP 2021. In addition, we added an option for bounding the maximum depth of symbolic exploration along a single path. This is called the search bound.

Concolic Executor. JDART and GDART share the same model and strategy for implementing the concolic executor. Both tools instrument a guest JVM written in JAVA that is executed on top of a normal JVM. Figure 3.5 shows this on the left. Without further modification, the program under analysis runs on top of the guest VM. the guest

VM, which interprets all bytecodes of the program under analysis. The host VM's only job is executing the guest VM. This separation is absolute. The program running on the guest VM cannot access any information in the heap space or on the stack of the host VM directly unless the guest VM transfers information from the program under analysis into the host VM and vice versa.

Consequently, it is possible to manipulate the heap space layout arbitrarily in the guest VM, e.g., for extending objects with additional fields carrying the symbolic information. Further, the stack in the guest VM can be patched for processing concrete values together with symbolic annotations. All bytecodes can be instrumented to generate symbolic information apart from the concrete execution. Furthermore, the class loader is controlled, allowing the implementations of classes or single methods to be replaced. Paper I [107] on JAINT explains in detail how JDART uses heap space manipulation, stack manipulation, and bytecode instrumentation. The same techniques are used in SPOUT, the concolic executor of GDART, for generating concolic traces. Paper VI [105] describes SPOUT's internals.

Intercepting Method Invocations for API Level Symbolic Encodings. In the version of JDART described in the tool paper at TACAS 2016 [89], symbolic encoding works for primitive types in the JVM. The bytecodes are instrumented for all primitive types to take care of the symbolic annotations. Objects are explored by tracking the effects on the primitive type components and ignoring the overall object structure. In order to lift the symbolic encoding for string operations to the JAVA standard library level—a process that I will discuss in more detail in the next chapter—this is insufficient. Instead, we need to intercept the invocation of the method on the string object, compute the concrete effects in the JVM, and track the symbolic encodings separately on the string operation level.

To better understand the problem, look at the code example in Listing 3.3 and the compiled bytecode for the same example in Listing 3.5. The bytecodes in Line 6 to Line 10 of Listing 3.5 are connected to the creation of the *StringBuilder* object $buffer$. As the string passed to the constructor of the $buffer$ object is generated using the NONDETSTRING method on the *Verifier* class, it is considered a nondeterministic string. The dynamic symbolic execution creates a symbolic variable, e.g., B for this string and tracks it as an addition to the object. Next, during the evaluation of the assertion condition, the CHARAT method is invoked on the $buffer$ object that contains the symbolic string B. This is done using the bytecode *invokevirtual* in Line 15 after pushing the object reference on the stack in Line 13 and the constant zero in Line 14. And this bytecode is where things get complicated. As I will explain in Chapter 4, we will encode this method on the string library level. This means the symbolic execution trace contains a named function that is interpreted for encoding the CHARAT method. At the same time, the concrete operation has to compute the concrete results. The computation of the concrete results invokes different methods on the STRINGBUFFER and STRING classes.

In the concrete example, the index value is at some point compared with the length of the string in the CHECKINDEX method shown in Listing 3.4. The problem is that the length in the $buffer$ object is a primitive field in the symbolic object that gets a

```
public static void main(String[] args) {
  StringBuilder buffer =
    new StringBuilder(
      Verifier.nondetString());
  assert buffer.charAt(0)
    == buffer.charAt(4);
}
```

Listing 3.3: The main method of the StringBuilderChars03 task in the SV-COMP 2022 Java track.

```
static void checkIndex(int index,
                       int length) {
  if (index < 0  index >= length) {
    throw new
    StringIndexOutOfBoundsException(
      "index " + index
      + ",length " + length);
  }
}
```

Listing 3.4: The CHECKINDEX method part of the STRING class in the standard JAVA library.

```
public static void main(String[]);                     1
  descriptor: ([Ljava/lang/String;)V                    2
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC                3
  Code:                                                 4
    stack=3, locals=2, args_size=1                      5
   0: new        #2                                     6
   3: dup                                               7
   4: invokestatic #3   // nondetString                 8
   7: invokespecial #4  // <init>                       9
  10: astore_1                                          10
  11: getstatic      #5                                 11
  14: ifne           38                                 12
  17: aload_1                                           13
  18: iconst_0                                          14
  19: invokevirtual #6  // charAt                       15
  22: aload_1                                           16
  23: iconst_4                                          17
  24: invokevirtual #6  // charAt                       18
  27: if_icmpeq      38                                 19
  30: new            #7                                 20
//AssertionError
  33: dup                                               21
  34: invokespecial #8  // <init>                       22
  37: athrow                                            23
  38: return                                            24
```

Listing 3.5: The bytecode for the method on the left.

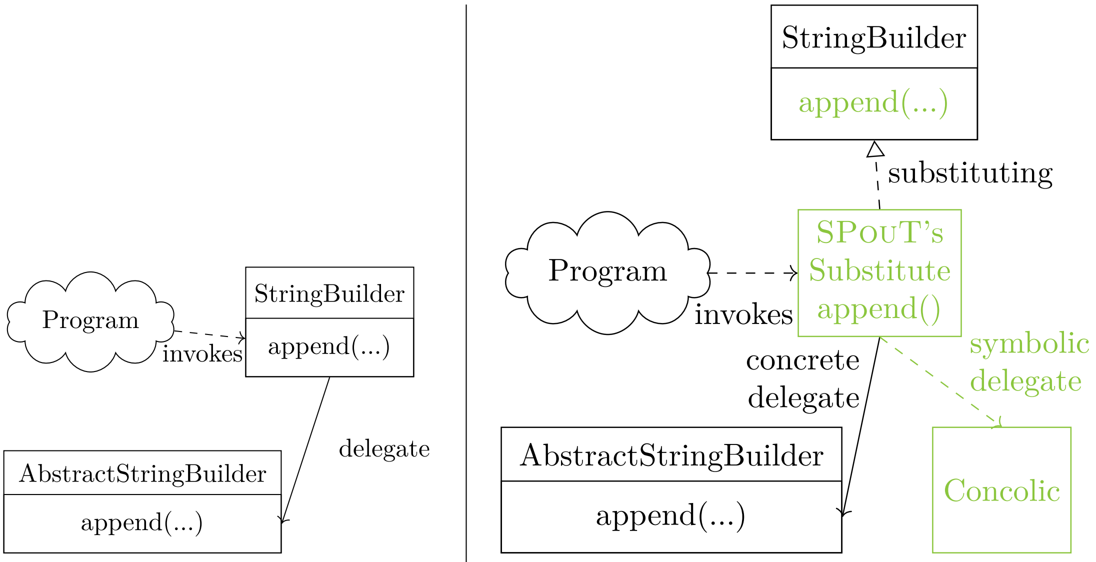
symbolic description. This symbolic description is required for various cases to correctly encode the effects of operations on the $buffer$ object. Therefore, the branching condition of the CHECKINDEX method becomes visible in the dynamic symbolic execution result. The primitive comparison bytecodes are instrumented to reflect this branching guard for the symbolically annotated length. But this time, this is not what we want. We are inside the concrete execution of the CHARAT method and the symbolic effects are encoded before or after the *invokevirtual* bytecode that actually triggers the execution of the CHARAT method. But none of the bytecodes executed to compute the concrete counterpart should alter the symbolic encoding before bytecode *aload_1* in Line 16 of Listing 3.5 is executed.

One way to work around this problem is to intercept all method invocations and guard those that are part of the symbolic instrumentation with special semantics. This can be done by altering how the JVM looks up invocation targets but slows down overall execution. Another possibility would be patching the code required for the symbolic annotations into the standard library, as in the SymbolicString approach by Shannon et al. [132]. But this either requires recompiling the standard JAVA library or patching the system under analysis to use the symbolic annotated classes. As the dynamic symbolic execution already instruments the JVM, changing any of the classes of the system under analysis or the runtime is the wrong solution in our case. JPF-VM and GRAALVM both support another mechanism that replaces an arbitrary method in any class by replacing it. The JPF-VM calls these *(Native-)Peers* and the GRAALVM calls them *substitutions*.

In this thesis, I will call these method *substitution methods* as common terminology. In both virtual machines, the consequence is that the original method body contained in the class file is not loaded, and instead, the substitution method is executed. This allows interception of the method invocation and patching it with symbolic annotations. The only problem is that the original method body is no longer executed, but the effects of the concrete execution have to be established in the heap of the JVM, allowing the concrete execution to continue without altering the semantics. Otherwise, it is no longer a concolic executor. In the following paragraphs, I present how delegation to the Host VM pattern and calling parent methods pattern helps to execute the concrete method semantically. Both patterns enable symbolic encoding of API level in the concolic executor and are important technical implementation details for supporting symbolic encodings on the Java library level.

Delegation to the host VM. Figure 3.5 describes on the right how delegation to the host VM for concrete execution works. In the example, the substitute method that replaces the charAt method converts the strings from the guest VM to the host VM. On the host VM, it calls the concrete implementation of charAt with the converted value and converts the result back to the guest VM. In the guest VM the substitution uses it as a return value of the substitute method. As the guest VM maintains the symbolic encoding, the computation inside the host VM does not influence the symbolic representations. This is possible for all cases where the host VM and the guest VM have semantically equivalent implementations. If the guest VM and the host VM differ in their semantics, e.g., because they operate on different language levels, this is impossible. Moreover, this delegation only works for pure functional methods without side effects leading to internal state changes of the object within the heap storage area of the guest VM. Otherwise, applying the delegation pattern to mask concrete execution in the symbolic encoding is impossible. Symbolic tracking is performed within the substitution method that operates in the context of the guest VM, ensuring that the symbolic annotations are attached to the final result when it reenters the guest VM and is available in future method calls. JDart and GDart use this strategy extensively to implement concrete behavior for stateless string functions. But the concepts work the same way in both VMs and conceal the VMs' switch from the program under analysis. This strategy is only possible so long as the tools use two layered VMs.

Invocation of Outsourced Implementations. For methods that change the state space in the heap of the guest VM, it is impossible to delegate concrete execution to the host VM. Replicating the changes in the heap space requires the implementation of a semantically equivalent concrete semantic in the guest VM. For many methods affected by this problem, the design of the concolic executor exploits the fact that the Java standard library bundles a common implementation in a transparent super class. Often, these super classes are package- or module-private and are not documented in the Java documentation as they are not intended to be used directly. But as the functionality is implemented in the super class and the externally exposed child class works as a facade to the super class, the concrete semantic on the heap is also established by the super class. The substitution method only has to replace calls to the other internal classes in



(a) The call chain in the Java standard library.

(b) The effects of substituting the *append()* method on the call chain.

Figure 3.6: Methods of replacing the concrete method in the guest VM, but allow to call other methods for reimplementing the original behavior.

order to establish the expected effects on the heap of the guest VM.

The left side of Figure 3.6 shows one such example, where *StringBuilder* and *StringBuffer* share the same parent class *AbstractStringBuilder* that does the actual implementation and heap space manipulation involved in invoking methods on objects of these types. As the substitute method is a replacement of the original method body, it inherits the call context of the original method, allowing the same package-private methods to be called. Therefore, the substitute method can replicate the call to the parent class to execute the concrete manipulation in the heap space. In addition, it adds the code for symbolic encoding. This way, it is often possible to replicate the concrete behavior of the substituted method body and the side effects on the heap state without rewriting all the changes to the heap in the substitute method. The implementation maintains the original semantics faithfully in the concrete execution. Using the facade pattern in standard libraries or frameworks, therefore, eases symbolic instrumentation for components that are encoded on a higher level than primitive types.

3.3 The Concrete vs. Symbolic Tradeoff

The purely symbolic execution of Java has to model every effect of the execution symbolically. If the target is an SMT-Lib encoding, the effects must be described in the SMT-Lib language. While this is easily possible with primitive type operations as they map directly into the bit vector or floating-point theory of SMT-Lib, the symbolic representation of reference type data structures (aka all objects in the JVM) is more challenging. Combining concrete and symbolic execution in dynamic symbolic execution allows balancing of the volumes of concrete vs. symbolic encoding. I will first explain how the concrete encoding restricts search space and allows horizontal scale-out of the analysis. Secondly, I will show how symbolic encoding allows environment modeling to overcome situations that normally require external input during concrete execution.

Concrete Drivers for Slicing the Search Space. A method under verification typically has some parameters or other source of external user input that allows a certain level of randomness during program execution. In verification slang, this is often called the nondeterministic part of the program compared to the deterministic part that is fully predictable. In the example of Listing 3.1, the X parameter to method *foo* is nondeterministic. Symbolic execution starts executing the method *foo* and treats X symbolically.

Running a program concretely in JVM requires starting the execution with the main method. It is impossible to start execution of *foo* without embedding it in the main method. JDart used some configuration mechanism to pick up the method under analysis and locate the symbolic parameters. This decouples the driver completely from the analysis configuration. For SV-COMP 2020 [102], we integrated the *Verifier* class of SV-COMP into the JDart tool so that values could be marked as nondeterministic if they were created by the *Verifier* class. It is always assumed that the value is at least initialized so that it is possible to create a variable of the corresponding data type in the SMT-Lib language. We used this *Verifier* class in the driver of the OWASP benchmark for the experiments with the Jaint framework in Paper I [107], as well for marking nondeterministic Strings in the analysis.

Modeling nondeterminism using the *Verifier* class is convenient, as the driver becomes the configuration for the analysis and combines both in a single file. But while the assumption that primitive data types are always initialized holds in Java, it is a different story for reference data types, e.g., strings. In Java, a string value might be a null pointer if it is not already initialized. Given the SMT-Lib language's semantics, it is impossible to encode in the SMT problem that a variable might be a null pointer. Together with the assumption that a value has to be initialized to be controlled symbolically, this means that the analysis will never execute paths that require the string variable to be a null pointer. Limiting the symbolic search space in the concrete driver makes the symbolic encoding of the remaining problem easier.

Next, I will demonstrate the limitation of the search space by example for the method foo in Listing 3.1. As a short reminder, all execution paths terminate successfully, except the one where foo is invoked with 94. A possible driver is the main method in Listing 3.6. If this driver is used, the problem will be explored as described in Section 3.1. However, there is no hard rule in the machinery—like invoking the method with a not null value—that

```
public static void main(String[] args){
  int i = Verifier.nondetInt();
  foo(i);
}
```

Listing 3.6: An unrestricted driver for method foo from Listing 3.1.

restricts the driver from applying restrictions on the search space.

For the foo example, the driver can also restrict the input to any value other than 94 as shown in Listing 3.7. In this case, dynamic symbolic execution will not find the assertion violation. On the other hand, a second driver might analyze only the path, where the nondeterministic value is 94. This way, the driver scopes the search space. i refer to this effect whenever I say the result of the analysis is correct within the search space limited by the driver method.

```
public static void main(String[] args){
  int i = Verifier.nondetInt();
  if (i != 94){
    foo(i);
  }
}
```

Listing 3.7: A driver for method foo from Listing 3.1 restricting the nondeterministic value.

For a full analysis of all possible paths in a program, some analyses will require more than one driver method. If objects are involved, splitting the search space in the set of paths where the object is null and a set with those paths where the object is not null is one possible split. The advantage is that it is possible to run the dynamic symbolic execution of the subspace reachable by a driver class independent from the other classes. Hence, this allows a full horizontal scale-out. By limiting the search spaces systematically in the concrete driver, the slicing produced might be finer than only null pointers vs. initialized values, as shown for the foo method in the paragraph above. The interplay between the concrete driver and the state space requiring symbolic exploration provides strength in scaling dynamic symbolic execution.

For the Jaint framework, we also limit in the concrete driver how many cookies the request passed to an invocation of an HttpServlet has. This cuts down the required symbolic expression power to model an arbitrary cookie array, but it also requires some reasoning about the implication of the limitations in the concrete driver on the verification verdict. This reasoning needs to explain why the scope defined by the symbolic driver is useful to answer the relevant security properties. Otherwise, there is the risk that the restrictions implied by the driver will mask a security weakness.

In the long run, combining the analysis from different dynamic symbolic execution runs into a final verdict, and cross-checking it by applying some branch coverage metric, would be desirable. However, this thesis leaves that cross-checking as an open question in favor of focusing initially on the Jaint framework first.

Environment Modelling Using the Substitution Pattern. Substitution methods and symbolic encodings allow not only symbolic modeling of library calls. Dynamic symbolic execution also uses these techniques to mock potentially asynchronous environments symbolically, cutting dependencies on other libraries. The concrete executor step in dynamic symbolic execution generally has the same challenge as any dynamic testing method, as all of them need to run the system. If the code under test waits asynchronously on an external event, it might block for a long time in the test until the external event is somehow triggered. I will not discuss general strategies for modeling the test environment in more detail here but I will demonstrate by example how substitution methods allow creation of what Feathers, in his book “Working Effectively with Legacy Code” [63], calls *fake-objects* for decoupling the execution of systems from the externally controlled state. In this case, symbolizing concrete parts allows analysis of the system behind the blocking concrete call. Moreover, as asynchronous calls in Java typically take input from somewhere, the symbolic modeling aligns well with the semantics of these methods that introduce new nondeterministic data to the program.

Consider the example of using a socket in Listing 3.8. A socket is first bound to a specific port in Line 2, and Line 7 later reads the incoming data from the socket using a *BufferedReader*. In between, the code creates a couple of different stream readers on the socket's input stream. The read operation is supposed to block in Java until the socket binds against the specified port and receives data over the network. For analyzing such programs, it is sometimes desirable to specify a substitute method that does not block and, e.g., returns a nondeterministic (or concolic) string value, or raises one of the potentially occurring errors. This technique allows the definition of arbitrary models and replacements for original Java code and makes it possible to inject symbolic models, skipping parts of the concrete implementation. It is used wherever analysis of a target program is impossible because of a blocking implementation.

```
...                                                   1
socket = new Socket(”host.example.org”, 1234);        2
readerInputStream = new InputStreamReader(            3
    socket.getInputStream(), ”UTF–8”);                4
readerBuffered = new BufferedReader(                  5
    readerInputStream);                               6
String stringNumber = readerBuffered.readLine();      7
...                                                   8
```

Listing 3.8: An example of socket use in Java.

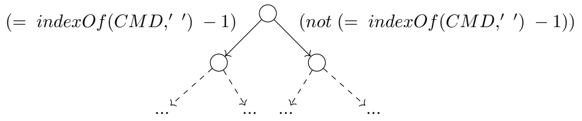
4 String Operation Encoding for DSE

A central contribution of this thesis is adding symbolic reasoning for JAVA programs with strings to the dynamic symbolic execution engines presented here. We briefly mentioned the string handling as contributions to JDART for SV-COMP shortly in Paper II [102] and Paper III [103] but never explained how the string encoding works. Paper II [102] only states that we use a bitvector-based string encoding, and Paper III [103] states that we switch from a bitvector-based encoding to a SMT-Lib string-based encoding. For GDART, Paper VI [105] explains briefly the specific details for the handling of strings in SPOUT. As none of these papers discusses the required scope for a symbolic encoding of the JAVA string library or explains the limitations carefully, I will complement those previous papers with a discussion of implementation scope, explain the difference between the two encoding strategies, and highlight the open challenges for dynamic symbolic execution of programs with strings that are pointed out in this thesis. But first, I will start with an example that will clarify the challenges of integrating symbolic strings into dynamic symbolic execution.

Motivating Example. In the way we program, strings are often used as the smallest unit of a complex data structure for storing textual information, similar to the usage of primitive data types. Consequently, a string library with operations on strings is a core component of most modern programming languages, and programmers are usually not interested in the details abstracted away by the string library. For example, in the case of JAVA, Eler et al. [62] have shown in a study across 147 programs that strings are the most often used data type after primitive integers and complex objects in JAVA programs. But strings are not a primitive data type for most hardware architectures. Hence a string value will be internally represented using numeric primitive types. This is also the case for the JVM. The concrete implementations change over time, but in JAVA 17, a string consists internally of a byte array, a string length, and some encoding information all represented as numeric data types. To analyze string operations, the analysis must decide between either encoding the primitive numeric representation, often called a bitvector encoding (cf., e.g., Redelinghuys et al. [120]), or intercepting the string operations and encoding them on the string data type level. Shannon et al. [132] and Bjørner et al. [26] have shown that encoding the analysis problem on the string theory level is beneficial for decision procedures and eases faithful error modeling in the analysis. This thesis will call this encoding strategy *string theory encoding.* For building a dynamic symbolic execution engine, it is therefore important, when deciding on the symbolic encoder, to encode string operations symbolically in either bitvector or string theory encoding. Figure 4.2 visualizes this design question. The concrete executors must establish the concrete execution result in the heap and on the stack in every execution. Depending

```
public void site_exec(String cmd) {                 1
  String r; // result                               2
  String p = ”/home/ftp/bin”;                       3
  int j, sp = cmd.indexOf(' ');                     4
  if(sp==-1){                                       5
    j = cmd.lastIndexOf('/');                       6
    r = cmd.substring(j); }else{                    7
    j = cmd.lastIndexOf('/', sp);                   8
    r = cmd.substring(j);                           9
  }                                                 10
  if (r.length() + p.length() > 32) {               11
    return; // buffer overflow                      12
  }                                                 13
  Stringbuf=p+r;                                    14
  if (buf.contains(”%n”)) {                         15
    throw new Exception(”THREAT”);                  16
  }                                                 17
  execute (buf);                                    18
}                                                   19
```

Listing 4.1: Example of a program with strings containing a code injection vulnerability taken from Redelinghuys et al. [120].



$(= \mathit{indexOf}(CMD,' ') \ -1)$
$(\mathit{not}\ (= \mathit{indexOf}(CMD,' ') \ -1))$
...
...
...
...

Figure 4.1: The string constraints get directly integrated into the constraint tree.

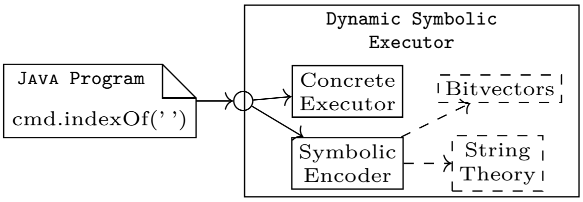


Figure 4.2: During the execution of the Java program, the dynamic symbolic executor splits execution into the concrete and the symbolic part. The question to solve is, what the symbolic encoding should look like: bitvectors or string theory.

on the configuration, the symbolic encoder maps the string operation to one of the two possible encodings. Separating concrete execution from the symbolic encoding allows this flexibility.

Next, I will give an example to demonstrate what string theory encoding looks like during dynamic symbolic execution and how string operations are encoded on the string theory level in the constraint tree. Listing 4.1 presents the *site_exec* method that takes a single string parameter *cmd* as input. For the purposes of analyzing this method, consider this parameter as a nondeterministic value; every string operation involving this parameter must then be traced in the symbolic execution tree. The constraint recorded in the symbolic execution tree is fine-grained enough to generate models using the constraint solver that drives the execution down the different paths. For example, the variable *sp* gets as symbolic value the INDEXOF(CMD, ' ') operation assigned in Line 4 and the result is compared against −1 in Line 5. Figure 4.1 shows how the INDEXOF(CMD, ' ') and the comparison of the result against the numeric value is expressed in the symbolic constraint tree on the string theory level. It does not decompose into bitvector constraints about the value array representing the string in the JVM as is the case for bitvector encoding.

Historically, JDart used to encode only primitive types in its original design, which does not allow constraints to be kept on the string theory level in the constraint tree. Redelinghuys et al. [120] claimed that a bitvector or a string theory encoding is less relevant for analyzing programs with strings, and the integration of string and numeric constraints is more important for the performance of the overall approach. Therefore, we started with a bitvector encoding in JDart. Later, we discovered the advantages of the string theory encoding and extended JDart with that encoding.

For the implementation of JDart and GDart, this raises three central questions regarding the general design of the component that deals with string operations in a dynamic symbolic execution engine:

SRQ1: Which functions are part of the string library that must be intercepted?

SRQ2: How are these functions encoded?

SRQ3: What are the limitations of the encoding?

In the following sections, I will define the scope of the Java string library as a set of Java data types and operations that require support in encoding for the analysis of Java programs. I will call this scope definition the language $\mathcal{SL}_{\mathcal{J}}$. $\mathcal{SL}_{\mathcal{J}}$ is my answer to SRQ1. I will then describe and summarize our experiences with encoding $\mathcal{SL}_{\mathcal{J}}$ using bitvectors. I call this the $\mathcal{SL}_{\mathcal{BV}}$ encoding strategy. The following subsection describes the $\mathcal{SL}_{\mathcal{SMT}}$ encoding strategy translating $\mathcal{SL}_{\mathcal{J}}$ into string theory constraints. The section concludes with a comparison of $\mathcal{SL}_{\mathcal{BV}}$ and $\mathcal{SL}_{\mathcal{SMT}}$, followed by a discussion of open challenges not answered in this thesis. All these sections together answer the string research questions SRQ2 and SRQ3. But before I continue with this, I want to highlight two specific solutions presented for SRQ1 and (partially) SRQ2 in the literature.

Symbolic Representations of Strings in the Literature. Two papers significantly influenced the design of the solution as presented in this thesis significantly: Bjørner et al. [26] and Redlinghuys et al. [120].

Bjørner et al. [26] described first how lifting the symbolic representation of string operations from a bitvector encoding to a higher abstraction language benefits dynamic symbolic execution engines. They introduced the string library language $\mathcal{LL}$ to model string operations in Pex [138] and mapped these to an SMT-Lib encoding before invoking Z3. By modeling string constraints separately from integer constraints, they could interweave the symbolic representation and concrete execution and raise exceptions. This allowed faithful analysis of all program paths, including exception handling.

Redelinghuys et al. [120] compared bitvector encoding with automata-based decision procedures for symbolic analysis of Java programs. Bitvector encoding uses the Z3 solver, while automata-based representations use the automaton solver part of the Java String Analyzer (JSA) project by Christensen et al. [41]. They introduced the language $\mathcal{SL}_{\mathcal{R}}$ as a minimum subset of operations required to analyze programs involving strings. The result of the comparison was that there are no clear benefits in using a bitvector vs. automata-based solver. Instead, the main challenge they pointed out was achieving integration between the constraints defining the string content part and those defining numeric operations on the string, e.g., limiting the string length.

In comparison with the string library $\mathcal{LL}$ presented by Bjørner et al. [26], the Java string language is more expressive, as I will show in detail in section 4.1. Hence it is undecidable in the general case as the $\mathcal{LL}$ subset of $\mathcal{SL}_{\mathcal{J}}$ is already undecidable in the general case. While this is generally an unwanted theoretical result for building tools using decision procedures for string theory, there are still decidable subparts and decision

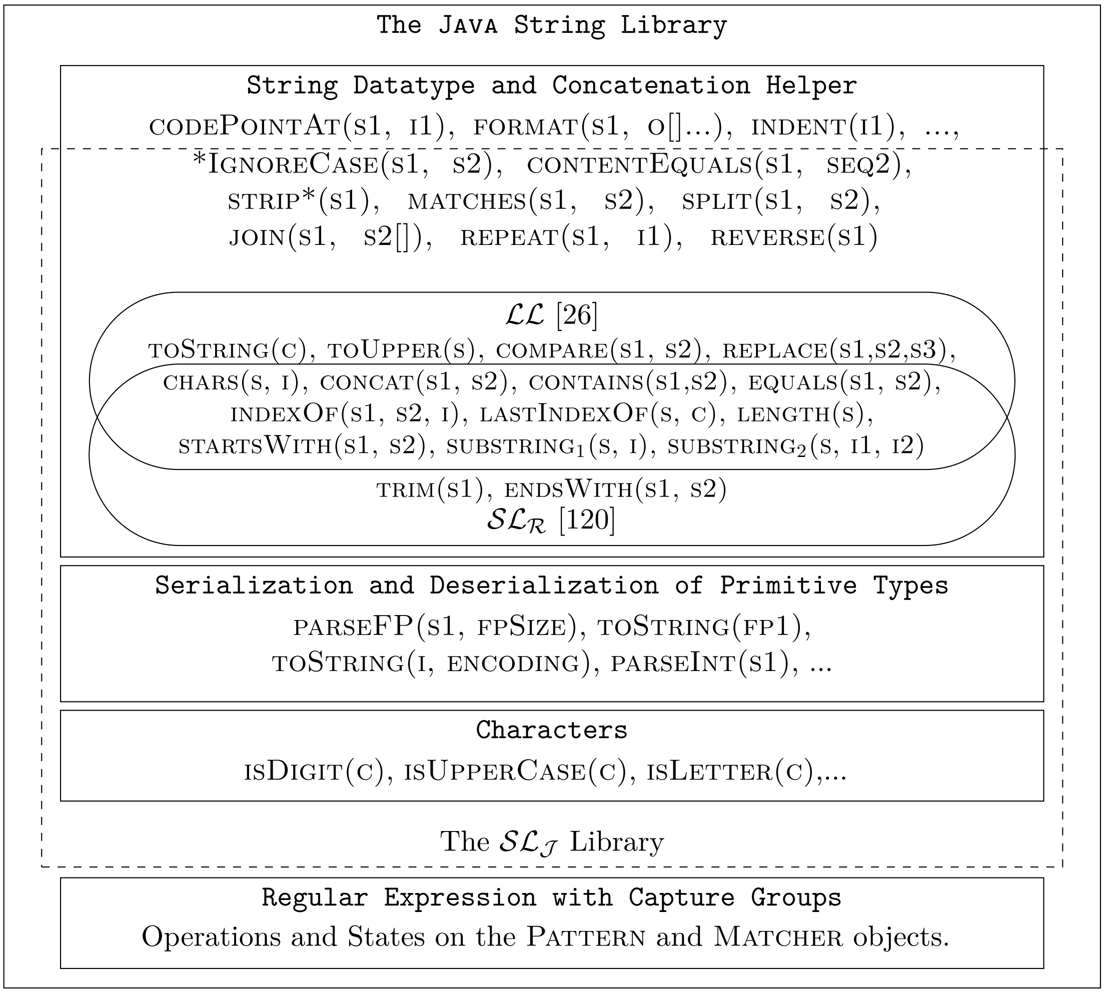

The Java String Library
String Datatype and Concatenation Helper
codePointAt(s1, i1), format(s1, o[]...), indent(i1), ...,
*IgnoreCase(s1, s2), contentEquals(s1, seq2),
strip*(s1), matches(s1, s2), split(s1, s2),
join(s1, s2[]), repeat(s1, i1), reverse(s1)
$\mathcal{LL}$ [26]
toString(c), toUpper(s), compare(s1, s2), replace(s1,s2,s3),
chars(s, i), concat(s1, s2), contains(s1,s2), equals(s1, s2),
indexOf(s1, s2, i), lastIndexOf(s, c), length(s),
startsWith(s1, s2), substring$_1$(s, i), substring$_2$(s, i1, i2)
trim(s1), endsWith(s1, s2)
$\mathcal{SL}_{\mathcal{R}}$ [120]
Serialization and Deserialization of Primitive Types
parseFP(s1, fpSize), toString(fp1),
toString(i, encoding), parseInt(s1), ...
Characters
isDigit(c), isUpperCase(c), isLetter(c),...
The $\mathcal{SL}_{\mathcal{J}}$ Library
Regular Expression with Capture Groups
Operations and States on the Pattern and Matcher objects.

Figure 4.3: The required symbolic operations in $\mathcal{SL}_{\mathcal{J}}$ for encoding the Java 17 string library.

heuristics that in practice solve many problems involving string constraints. From the tool building and engineering perspective, it is, therefore, crucial to understand the limits of string theory solving in available SMT solvers (cf. Paper IV [100], which contributes to answering the performance question on existing benchmarks) as well as the limits in the expressiveness of the SMT-Lib language.

4.1 $\mathcal{SL}_{\mathcal{J}}$ a Tailored Subset of the Java String Library for DSE

The Java string library is split across multiple classes in the Java standard library. It consists of five main parts: the string data type itself (*java.lang.String*), classes for string concatenation (e.g., *java.lang.StringBuilder*), the boxing objects for other primitive types containing the code for type conversion and value parsing from a string representation

Method	SF110	SV-COMP	Jenkins	$\mathcal{SL}_\mathcal{J}$	$\mathcal{LL}$	$\mathcal{SL}_\mathcal{R}$	$\mathcal{SL}_\mathcal{BV}$	$\mathcal{SL}_\mathcal{SMT}$
java.lang.String								
equals	7 819	103	333	✓	✓	✓	✓	✓
length	4 329	47	232	✓	✓	✓	✓	✓
substring	2 108	21	189	✓	✓	✓	✓	✓
trim	1 281	23	94	✓		✓		
indexOf	1 265	15	72	✓	✓	✓		✓
format	1 114	0	156					
startsWith	1 001	8	147	✓	✓	✓	✓	✓
charAt	993	71	54	✓	✓	✓	✓	✓
equalsIgnoreCase ()	752	4	24	✓				(✓)
endsWith	629	3	70	✓		✓		✓
compareTo	618	6	19	✓	✓			(✓)
getBytes	511	5	76					
replaceAll	507	9	23	✓	✓			✓
valueOf	497	30	20	✓				
split	461	5	56	✓			(✓)	
replace	427	5	95	✓	✓			✓
lastIndexOf	425	8	26	✓	✓	✓		
toUpperCase ()	343	3	24	✓	✓			(✓)
contains	283	7	104	✓	✓	✓	✓	✓
toCharArray	176	1	5					
compareToIgnoreCase ()	170	0	10	✓				(✓)
concat	76	7	0	✓	✓	✓	✓	✓
matches	51	14	28	✓				✓
replaceFirst	39	3	8	✓				✓
toLowerCase ()	25	18	39	✓				(✓)
regionMatches	12	4	0	✓	✓	✓	✓	✓
getChars	10	1	0					
subSequence	4	0	0					
contentEquals	2	0	0	✓	✓	✓	✓	✓
join	0	0	21	✓				
java.lang.Float								
parseFloat	98	19	2	✓			(✓)	(✓)
valueOf	50	0	0	✓				
toString	37	0	0	✓				
java.lang.Double								
parseDouble	247	1	2	✓			(✓)	(✓)
valueOf	75	2	0	✓				
toString	40	2	0	✓				
java.lang.Character								
isWhitespace	59	1	0	✓				
toString	56	0	0					
charValue	51	0	0					
isDigit	46	4	2	✓				✓
toUpperCase ()	38	2	1	✓	✓			(✓)
valueOf	38	0	0	✓				
toLowerCase ()	25	2	0	✓				(✓)
isLetter	17	5	2	✓				
isUpperCase	12	1	0	✓				
isSpaceChar	8	0	0	✓				
digit	5	1	0					
isLetterOrDigit	5	1	0	✓				
equals	4	1	0	✓				✓
isJavaIdetifierStart	4	1	0	✓				
isJavaIdentifierPart	4	1	0	✓				
toChars	3	0	0					
getNumericValue	2	0	0	✓				
getType	2	0	0					
isLowerCase	2	1	0	✓				
forDigit	1	2	0					
isISOControl	1	0	2	✓				
isUnicodeIdentifierPart	1	0	0	✓				
isUnicodeIdentifierStart	1	0	0	✓				
isDefined	0	2	0	✓				
java.lang.Integer								
parseInt	1 398	4	73	✓			(✓)	(✓)
valueOf	608	2	2	✓				
toString	444	1	17	✓				
toHexString	54	0	3					
decode	9	1	1	✓				
toBinaryString	4	0	0					
toOctaleString	1	0	0					
java.util.regex.Pattern								
matcher	390	2	64					
compile	383	2	64					
pattern	20	0	0					
split	13	0	1					
quote	6	0	0					
matches	2	0	1					
java.lang.StringBuilder								
append	4 647	43	322	✓	✓	✓	✓	✓
toString	689	17	78	✓	✓	✓	✓	✓
delete	8	2	0	✓				
replace	7	0	0	✓	✓			✓
indexOf	6	0	0	✓	✓	✓		✓
insert	4	3	5	✓				✓
lastIndexOf	3	0	0	✓	✓	✓		
deleteCharAt	2	2	0	✓				
reverse	0	2	0					
java.util.regex.Matcher								
group	370	6	38					
find	201	3	16					
matches	191	0	43					
start	21	0	6					
end	19	0	7					
appendReplacement	13	0	1					
replaceAll	12	0	2					
groupCount	10	0	4					
appendTail	8	0	1					
quoteReplacement	5	0	0					
replaceFirst	1	0	3					
java.lang.StringBuffer								
append	0	82	1	✓	✓	✓	✓	✓
charAt	0	0	1	✓	✓	✓	✓	✓
~~*codePointAt*~~	0	0	0					
~~*compareTo*~~	0	0	0	✓				(✓)
~~*delete*~~	0	0	0	✓				
~~*deleteCharAt*~~	0	0	0	✓				
~~*indexOf*~~	0	0	0	✓	✓	✓		✓
insert	0	1	0	✓				✓
~~*lastIndexOf*~~	0	0	0	✓	✓	✓		
length	0	4	2	✓	✓	✓	✓	✓
~~*replace*~~	0	0	0	✓	✓			✓
~~*reverse*~~	0	0	0					
~~*setCharAt*~~	0	0	0					
~~*substring*~~	0	0	0	✓	✓	✓		✓
toString	0	24	3	✓	✓	✓	✓	✓
setLength	0	1	1					

Table 4.1: A comparison of usages of different operations from the JAVA 17 string library in three different sets of JAVA classes. SF110 represents 105 of the java programs included in the EvoSuite [66] used for unit test case generation benchmarking. The analysis failed to analyze all dependencies for five of the 110 projects. SV-COMP is the collection of all JAVA classes used in the SV-COMP 2022 task set. Jenkins describes all classes and dependencies used in the core Jenkins package. Red are these operations that are used in the programs but are not part of the SV-COMP challenge. The crossed out functions in the *StringBuffer* are defined in the library but never used by a program. The ticks in the $\mathcal{SL}_\mathcal{J}$, $\mathcal{LL}$, $\mathcal{SL}_\mathcal{R}$, $\mathcal{SL}_\mathcal{BV}$, and $\mathcal{SL}_\mathcal{SMT}$ columns show whether this operation is included in the language or encoding. Braces around the tick imply only partial support in the encoding for this operation or that it is only contained in JDART.

(e.g., *java.lang.Float*), the character boxing object for dealing with the conversion between the numeric and string semantic of a single character (*java.lang.Character*), and pattern matching related operations (*java.util.regex.Matcher* and *java.util.regex.Pattern*).

To answer SRQ1, I analyzed the usage of the Java string library in three different projects: The EvoSuite, SV-COMP 2022, and Jenkins. The EvoSuite [66] consists of 110 Java programs collected from SourceForge and has been published to demonstrate the efficiency of automated tools for unit test case generation. The table includes 105 of these programs, as the analysis failed for 5 of them while resolving the dependencies. The SV-COMP 2022 task set is the Java track of SV-COMP 2022 [18]. Jenkins is a single larger Java application. As Jenkins is a modern CI/CD Server, the codebase represents a full-grown Java application.

Next, I will briefly discuss why I collected these three sets of Java class files for analysis. For SF110, there is, from time to time, a discussion about whether the selection is representative for Java programs and of high quality or not (cf. Dimjašević et al. [58] complaining about empty benchmarks in the task set and Eler et al [62] comparing it with the R47 Java program set estimating that within both program sets 75% of all methods have only one execution path without branching, making large parts of the programs less attractive for symbolic analysis). But, to the best of my knowledge, it has not been shown that the SF110 is unrepresentative compared with other mixes of Java programs or real-world software in general. On the other hand, Eler et al. [62] show that it is comparable with the R47 benchmark set. However, it is a defined set of Java programs used in different papers to compare tool performance, so it is the first step toward a standardized benchmark. SV-COMP aims to establish a set of tasks that allows a representative comparison of different software verification tools. For this reason I have included it in the comparison. I included Jenkins, as it is a mature real-world application allowing comparison of the benchmark sets established by researchers with a larger production-grade code base. Apart from that, the SF110 benchmark was released in 2014, and SV-COMP has many participants with Java 8 tools, so most of the newer changes introduced after the introduction of compact strings in JEP 254[1] are not contained in this benchmark, given that JEP 254 is a part of Java 9. As Jenkins had adapted some of these already (e.g., the join operation), I added it to the comparison. At the same time, as Jenkins is a well-maintained web server, it also shows that not all features of the Java library are used in full-sized Java programs.

Table 4.1 presents the analysis results. Methods of Java 17 not presented in the table are not used in these three projects. Based on the methods used and existing work in the literature, I define the scope of $\mathcal{SL}_{\mathcal{J}}$ here to include a desirable subset of Java string library operations for running the dynamic symbolic execution of programs with strings. The resulting scope of $\mathcal{SL}_{\mathcal{J}}$ is presented in Figure 4.3 and represents my answer to SRQ1. We see in the top functions defined in the Java string library part of the *String* data type and the concatenation helper classes *StringBuffer* and *StringBuilder*. Below, we see the functions involved in the serialization and deserialization of primitive types and characters. The part of these four groups that is included in $\mathcal{SL}_{\mathcal{J}}$ is within the dashed

[1] `https://openjdk.java.net/jeps/254` {last accessed: February 2022}

rectangle that shows the border of $\mathcal{SL}_{\mathcal{J}}$. The bottom shows regular expressions with capture groups, an important part of the Java string library that this thesis excludes from $\mathcal{SL}_{\mathcal{J}}$. These different parts are discussed in further detail below.

The String Datatypes and Concatenation Helpers. The string data type *String* and the concatenation helper classes *StringBuffer* and *StringBuilder* represent the first two of the five main parts identified previously. As shown in Table 4.1, the previously mentioned string languages, $\mathcal{LL}$ and $\mathcal{SL}_{\mathcal{R}}$, represent for the most part symbolic operations that are attributed to this part of the JAVA string library. Moreover, they partially overlap, as shown in Figure 4.3.

I define both string languages as a subpart of $\mathcal{SL}_{\mathcal{J}}$ as they represent often used methods. This includes equals, length, substring, indexOf, startsWith, charAt, lastIndexOf, contains, concat, and regionMatches. contentEquals is not discussed as part of $\mathcal{LL}$ or $\mathcal{SL}_{\mathcal{R}}$, but the symbolic encoding works the same as for equals, if the other object is encoded as a symbolic string value which is semantically required by the Java semantic. Therefore, mark it as part of the other two languages as well, and map it to the equals encoding internally.

The concatenation Helpers in Table 4.1 are the *StringBuffer* and *StringBuilder* class. Both share the same semantic for the API. Further, many aspects of their APIs match the *String* data type. With the symbolic expression of concat for two string variables, it is possible to model the append method of the buffer and builder API. The same applies for charAt, indexOf, lastIndexOf, and length methods. As each symbolic *StringBuffer* or *StringBuilder* is modeled using symbolic expressions, toString on these objects is implemented by returning the concrete string annotated with the current symbolic description of the builder or buffer. This does not require any special support in the symbolic encoding and is therefore always supported.

There are some additional methods supported exclusively in $\mathcal{LL}$ or $\mathcal{SL}_{\mathcal{R}}$. $\mathcal{LL}$ supports toUpper, compare, and replace which I define to be part of $\mathcal{SL}_{\mathcal{J}}$ as well. In consequence, $\mathcal{SL}_{\mathcal{J}}$ and $\mathcal{LL}$ overlap in encoding compareTo, replaceAll, replace, and toUpperCase on the *String* data type and toUpperCase on the *Character* data type. $\mathcal{SL}_{\mathcal{R}}$ supports endsWith and trim overlapping with $\mathcal{SL}_{\mathcal{J}}$ on these methods.

Next, I added a set of methods to $\mathcal{SL}_{\mathcal{J}}$ that is used in Java programs and might have influence on the control flow path, but is not expressible using the currently existing symbolic method encodings in $\mathcal{LL}$ or $\mathcal{SL}_{\mathcal{R}}$. For the *String* data type, these are equalsIgnoreCase, compareToIgnoreCase, matches, replaceFirst, toLowerCase, split, and join methods. Especially equalsIgnoreCase and compareToIgnoreCase are still often used, given the analysis of Table 4.1, and have the potential to influence branching conditions. So they are an important part of $\mathcal{SL}_{\mathcal{J}}$ for the analysis of the target domain. delete and deleteCharAt are the reverse method of insert on a *StringBuilder* or a *StringBuffer*. They are included in $\mathcal{SL}_{\mathcal{J}}$, as they are sometimes required; but they are also not the most important methods for the dynamic symbolic execution of programs with strings, as they are still rarely used.

At the same time, I excluded certain standard operations like format, getBytes, toCharArray, subSequence and codePointAt that (for different reasons) are challenging in their symbolic encoding. format is very handy for printing strings, but the encoding is not straightforward and describing symbolically the serialization and deserialization of objects to strings is currently difficult. A scaling solution requires the encoding of serialization and deserialization for all primitive datatypes and potentially unbounded collections. Table 4.1 shows that format is one of the most often used methods in the String data type. In the long term, expanding $\mathcal{SL}_{\mathcal{J}}$ in this direction is desirable, but as this requires major work on a theory solver that deals with unbounded collections, which is beyond the scope of this thesis, I have excluded it from $\mathcal{SL}_{\mathcal{J}}$ for the time being. codePointAt is introduced as a long-term replacement of charAt, but I have not found a single occurrence of this method in the programs analyzed and have therefore excluded it. Therefore, I exclude it. getBytes, toCharArray, and subSequence require encoding of arrays of characters or bytes symbolically and link the bytes with the string values. SMT-Lib does not support the encoding of these operations while maintaining the link between the chars of the string and the string value. As this requires additional support in the symbolic theory and is not on my main path of research for combining dynamic symbolic execution with taint analysis, I have defined these methods as out of scope for $\mathcal{SL}_{\mathcal{J}}$. I have excluded reverse on a *StringBuilder* or *StringBuffer* from $\mathcal{SL}_{\mathcal{J}}$, as it is only ever used in the SV-COMP competition and never in any program.

Primitive Type to String Serialization and Deserialization. Functions are also required for parsing floating-point values from strings, converting floating-point values to strings, representing different integers in different binary notations (most common are hexadecimal, octal, and binary encodings), and converting back from such a string encoding to an integer. . I consider these functions part of the string library, but it is generally debatable whether they belong to the library or to number representation. To the best of my knowledge, this has not yet been discussed in the literature. Nonetheless, full dynamic symbolic execution requires support for these conversions in the symbolic encoding and I, therefore, define parseFP(s1, fpSize), toString(fp1), toString(i, encoding), parseInt(s1), and valueOf(...) as part of $\mathcal{SL}_{\mathcal{J}}$. As valueOf is defined for number serialization in the *String* class and for deserialization in different number classes, e.g., the *Double* class. I only use three dots here for the parameter. In total, the parseFP, parseInt and valueOf methods are invoked in 2973 places in the SF110 class set of Table 4.2, making type conversion more important than the substring method and the fourth most often used feature of $\mathcal{SL}_{\mathcal{J}}$ in this group of analyzed classes. For Jenkins, this subclass counts 99 invocations, making type conversion the fifth most often used feature of $\mathcal{SL}_{\mathcal{J}}$.

Characters. The part of the string library dealing with characters needs methods for checking and applying domain restrictions for individual characters. There are many Boolean condition checks like isDigit(c), isUpperCase(c), isWhitespace(c), etc. that I have included in $\mathcal{SL}_{\mathcal{J}}$, as they have the potential to influence branching conditions. As these methods check for different source domain ranges for the parameter character passed in their semantic, they all have a similar structure as regards the nature of the required check. The semantics of the checks align with some of the different character classes supported in the pattern language in Java like: \p{Lower}, \p{Upper}, and

\p{Space}. These classes are only lower, upper, or white space letters. Support for encoding these character classes in the reasoning backend eases the modeling of Java string constraints.

At the same time, Table 4.1 shows that the methods defined for the *Character* class are seldom used (e.g. by Jenkins) in comparison with other parts of the Java string library. Depending on the analysis target, some dynamic symbolic execution applications will not need them for successful analysis. Nevertheless, they are part of $\mathcal{SL}_{\mathcal{J}}$.

Regular Expressions. Java's pattern language leads us to the fifth major part of the Java string library: regular expressions and pattern matching. The simplest way of matching a string $s1$ against a pattern $s2$ is to check the Boolean value of the match in the matches(s1, s2) operation[2] already offered by the standard string data type. However, Patterns are so important for the Java string library that they have their own class to represent them: *Pattern.* A *Pattern* object is built around a regular expression represented in the pattern object and compiled into an optimized form on creation. A pattern object allows strings to be split on every match of the pattern, as with the previously mentioned split(s1, s2) function on *String.* Moreover, it can check whether a string value matches the pattern.

A specific feature of the Java regular expression language is capture groups that allow back references. For Example, the following Java pattern "[s|S]ecurity (.*)$" matches everything in a line after the word "security" and a whitespace until the end of the line. Given the string "security for the win", the pattern matches. In contrast to pure regular expressions that either match or do not match, regular expressions with back references allow one to access the matched content after the match. The Java regular expression language uses parentheses for defining a capture group that allows access to all characters within this part of the pattern after a match. For the example string used, the content of the capture group is "for the win". Matching regular expressions with back references is an NP-hard problem [3]. As it is more common in the Java context to call these capture groups rather than back references, I will use the term capture groups in the remainder of this document.

The Java standard library uses the *Matcher* data type for working with capture groups in regular expressions. It allows a pattern to be checked for potential matches on a string and makes the capture groups accessible if a pattern is successfully matched. This class can replace or extract capture groups depending on the use case. Table 4.1 shows 370 calls to the group method on the *Matcher* class in the SF110 benchmark suite and 38 calls in the Jenkins application. They are accompanied by 191 invocations of matches in the SF110 benchmark suite and 43 invocations ofmatches in Jenkins on the *Matcher* class. In contrast, SF110 calls in 51 cases matches on the *String* class and Jenkins in 28 cases. Given these numbers, regular expressions with capture groups are more often used than regular expressions alone in the Java programs analyzed.

[2] *Remark:* While having a similar name, the regionMatches(s1, s2) operation has a totally different behavior, as it extracts two subparts from $s1$ and $s2$, checking them for equality. It could be modeled by compiling regionMatches into a sequence of substring and equals operations on the strings instead. This does not involve regular expressions. Therefore, I do not discuss this operation in more detail here, but we support it with the symbolic operations contained in $\mathcal{SL}_{\mathcal{J}}$

```
public static void main(String[] args) {       1
  args = new String[2];                        2
  args[0] = Verifier.nondetString();           3
  args[1] = Verifier.nondetString();           4
  String s1 = args[0];                         5
  String s2 = args[1];                         6
  assert s1.equals(args[0] + ” ”);             7
  assert s2.equals(args[1]);                   8
}                                              9
```

Listing 4.2: The StringConcatenation02 example from the SV-COMP Java track demonstrating concatenation and equality comparison of partially symbolic strings.

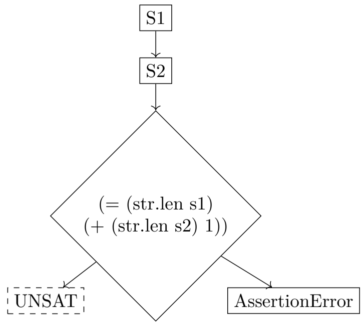


Figure 4.4: The symbolic constraint tree resulting from the analysis of Listing 4.2

Symbolic modeling of capture groups is currently not possible given the current capabilities of the string theory in SMT-Lib 2.6. Especially, modeling the extraction of a group from a string that follows a regular expression with capture groups is not expressible in SMT-Lib at the moment. Describing capture groups using bitvectors is also not possible. As there is no established way for encoding these operations and all their implications, I exclude these from $\mathcal{SL}_{\mathcal{J}}$ for now. Instead, I will discuss the open challenges in Section 4.4, as this is a limitation of $\mathcal{SL}_{\mathcal{J}}$ for the analysis of arbitrary Java programs. To make this visible, I have represented them at the bottom of Figure 4.3 outside of the dashed area representing $\mathcal{SL}_{\mathcal{J}}$.

4.2 JDart's Bitvector Encoding $\mathcal{SL}_{\mathcal{BV}}$

In this subsection, I present the scope and sketch the technical realization of the bitvector encoding $\mathcal{SL}_{\mathcal{BV}}$ developed as first string operation support in JDart (cf. Paper III [102] and Paper I [107]). The bitvector encoding aims to model the operations on strings as closely as possible to the representation of strings in the Java standard library.

In the concrete implementation, a string is a tuple of a length and an array of values representing characters. Similarly, a symbolic string s_{bv} in the bitvector encoding as presented here is a tuple $S_{bv} = (sl, sc[])$ where sl is a symbolic 32-bit bitvector representing the length of the string and $sc[]$ is an array of expressions representing the content of every single character in the string. To represent the char type of the JVM with some accuracy, this encoding uses an unsigned 16-bit bitvector to encode a single character. In this encoding, every character has its symbolic variable in the resulting SMT problem.

Example. Consider the example for an analysis task in Listing 4.2. Assume we execute it for the first time using dynamic symbolic execution. Line 3 and Line 4 create two fresh symbolic variables with a string length of zero as they are empty strings in our analysis by default in the first run: $S_{bv1} = (int_0, [])$ and $S_{bv2} = (int_1, [])$. Next, Line 7 executes a concatenation of S_{bv1} and whitespace. The symbolic expression that traverses through the JVM annotated to the result is $S_{concat} = ((+\ int_0\ 1), [' '])$[3]. This encodes the statement that the string has a symbolic prefix of length zero influenced by int_0 extended by the length of the concrete string concatenated with the symbolic string. The resulting concrete string is a single whitespace “ ”. A string of length zero does not have any chars. Therefore, the resulting string has a symbolic length, but no symbolic chars in this run. Next, the assertion in Line 7 is evaluated implying the symbolic check $(=\ S_{bv1}\ S_{concat})$. A two-step approach is used to encode this symbolic check into SMT-Lib using the bitvector encoding. First, the encoding generates constraints in the bitvector theory for checking the string length constraints as the numeric component of the check. Second, the actual string content is checked as to whether the numeric part is satisfiable. In this specific case, the numeric aspects of equals require that both strings are of equal size. The first step generates the constraint $(not\ (=\ int_0\ (+\ int_0\ 1)))$ and adds it to the symbolic decision tree as shown in Figure 4.4. As the concrete execution raises an assertion error, no further constraints are added and the branch terminates. As the negated constraint $(=\ int_0\ (+\ int_0\ 1))$ is unsatisfiable, this is the only path that ever gets executed in this example. There is never a semantic equivalent check on the symbolic characters is never required.

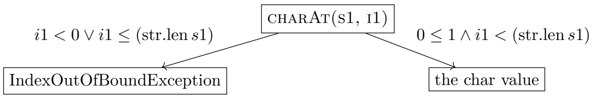

charAt(s1, i1)
$i1 < 0 \vee i1 \leq (\text{str.len}\, s1)$
$0 \leq 1 \wedge i1 < (\text{str.len}\, s1)$
IndexOutOfBoundException
the char value

Figure 4.5: The thrown exceptions for the charAt method depending on the numeric constraints involved.

Faithful Error Handling. The above-described two-step strategy follows the proposal by Bjørner et al. [26] for encoding string constraints during the analysis in PEX [138]. All encodings proposed for JDart use this two-step strategy for encoding string constraints. The advantage is that it is possible to model thrown errors faithfully, allowing the exploration of exception handling code during dynamic symbolic execution. For example, whenever the index of the charAt(s1, i1) method is out of bound, the $\mathcal{SL}_{\mathcal{J}}$ requires an IndexOutOfBoundException. Otherwise, it returns the character at position i1 in string s1 (cf.Figure 4.5). Separating the string length part of the problem from the string content representations allows the encoding of an index violation symbolically before throwing an exception during execution of the concrete method. Irrespective of the concrete theory used for encoding, we too—like Bjørner et al. [26]—require this separated modeling of the error condition for generating concrete inputs from the SMT model that drives execution down the error paths.

We require this separated modeling of the error condition for generating concrete inputs from the SMT model that drives the execution down the error paths as well, similar to Bjørner et al. [26] independent of the concrete theory used for the encoding.

$\mathcal{SL}_{\mathcal{BV}}$ covers the following operations symbolically: charAt(s1, i1), concat(s1, s2), contains(s1, s2), equals(s1, s2), length(s1), regionMatches(s1, s2), starts-

[3] *Remark:* As these values are represented as bitvectors in SMT-Lib, the real constraint annotated to the symbolic value is $(bvadd\ int_0\ \#x0000001)$. For readability, I keep the integer notation even for bitvectors in the main text as it is easier to convey the relevant ideas.

With(s1, s2), substring(s1, n1), and substring(s1, n1, k1).

Completely Encoded Methods. All in all, $\mathcal{SL}_{\mathcal{BV}}$ encodes only eight methods fully symbolically using the bitvector theory of SMT-Lib: startsWith, charAt, equals, concat, lastIndexOf, regionMatches, contains, and substring. In direct comparison with Table 4.1, , the number of ticks is higher than just these eight methods. The reason is that some methods map on the same symbolic encoding from different methods in the Java string library: e.g., append of the *StringBuilder* or *StringBuffer* class maps on the same symbolic concatenation operation in SMT-Lib as the *String* concatenation method. equals and contentEquals are mapped to the same symbolic equality between two string theory variables. length on *String*, *StringBuffer*, and *StringBuilder* all map on the same logical encoding for the length of a symbolic string variable. But we only implement it for *Strings*. Other methods, e.g., toString on the *StringBuilder* or *StringBuffer* do not require symbolic modeling, as the content of the builder or buffer is already encoded in the symbolic variable associated with the concrete object. However, as *StringBuilder* was the only concatenation helper class used in SV-COMP 2020 and we have used the $\mathcal{SL}_{\mathcal{BV}}$ encoding for this year, operations on *StringBuffer* are not implemented as part of JDart's $\mathcal{SL}_{\mathcal{BV}}$ implementation. In theory, they are encoded the same way as the methods invoked on *StringBuilder* and I have therefore ticked them in Table 4.1, but the interceptions are not in place in the code base.

Partial Encodings. There are four ticks in brackets in Table 4.1. One is related to splitting a string, and the other three are related to number de- and serialization. As described in the caption of Table 4.1, the brackets imply a partial or JDart-specific encoding. As these methods are only partially encoded, due to the challenges involved, I will discuss them in greater detail inwwww Section 4.4.

Brief Discussion. All in all, of the 72 methods ticked in $\mathcal{SL}_{\mathcal{J}}$ in Table 4.1, $\mathcal{SL}_{\mathcal{BV}}$ encodes 15 methods completely, using the previously described 8 symbolic methods, and 4 partially, using prototypes. As it requires significant effort to rebuild semantics for matching regular expressions in this encoding, $\mathcal{SL}_{\mathcal{BV}}$ has never supported operations like matches(s1, s2). Moreover, our implementation does not support direct comparison of a character against a certain class of characters, e.g., as required to encode the isUpperCase method. . This limitation alone is responsible for 18 (i.e. roughly a third) of the unsupported 53 methods. Finally, $\mathcal{SL}_{\mathcal{BV}}$ does not encode regular expressions in any form at the moment, limiting its applicability for methods requiring regular expressions as matches.

4.3 The String Theory Encoding $\mathcal{SL}_{\mathcal{SMT}}$

The string theory encoding used in JDart works similarly to bitvector encoding with the main exception that the string part of an operation is expressed in the string theory of SMT-Lib rather than encoding a string character-wise in the form of bitvectors. Paper II [103] calls this encoding strategy SMT-Lib encoding; but as bitvectors are also expressed using SMT-Lib, the name is misleading. String theory encoding is a better

$\mathcal{SL}_{\mathcal{J}}$ Operation	$\mathcal{SL}_{\mathcal{SMT}}$ Operation	$\mathcal{SL}_{\mathcal{J}}$ Operation	$\mathcal{SL}_{\mathcal{SMT}}$ Operation
concat(s1, s2)	$(\text{str.++}\ s1\ s2)$	isDigit(c1)	$(str.is\_digit\ c1)$
contains(s1, s2)	$(str.contains\ s1\ s2)$	length(s1)	$(str.len\ s1)$
contentEquals(s1, seq2)	$(=\ s1\ seq2)$	replace(s1, s2, s3)	$(str.replace\_all\ s1\ s2\ s3)$
endsWith(s1, s2)	$(str.suffixof\ s1\ s2)$	startsWith(s1, s2)	$(str.prefixof\ s2\ s1)$
equals(s1, s2)	$(=\ s1\ s2)$	substring(s1, i1)	$(str.substr\ s1\ i1\ (str.lens1))$
indexOf(s1, s2)	$(str.indexof\ s1\ s2\ 0)$	substring(s1, i1, i2)	$(str.substr\ s1\ i1\ (-\ i2\ i1))$
indexOf(s1, s2, i1).	$(str.indexof\ s1\ s2\ i1)$		

Table 4.2: Mapping from the $\mathcal{SL}_{\mathcal{J}}$ language into the SMT-Lib language (adapted from Table 1 in Paper VI [105]).

name, as the strategy uses the string theory in SMT-Lib. I will explain this encoding and discuss the advantages and disadvantages of the approach below. Paper VI [105] already described parts of this encoding in the context of SPouT.

String theory encoding extends on the idea of intercepting the Java string library and encoding the problem on the library level. This allows string operations to be mapped on the string data type in SMT-Lib and string operations to be used to express $\mathcal{SL}_{\mathcal{J}}$ in SMT-Lib. The encoding is unaware of the internal representation of a string. A symbolic string S_{smt} is converted to an SMT-Lib variable of type *string* representing the string object. The length is accessed using the SMT-Lib operation $(str.len\ S_{smt})$. This is similar to the usage of a string variable in a Java program. As described in Section 2.2, the numeric operations part of SMT-Lib returns integer values in SMT-Lib theory. The primitive integer Java data type `int` is a 32-bit bitvector datatype in the semantic, and we encode it in JDart and GDart in the bitvector theory. Hence, the integer values must be cast into bitvectors with the $(nat2bv)$ operation.

Reconsider, then, the example in Listing 4.2. In the SMT-Lib encoding, the newly introduced symbolic strings in Line 3 and Line 4 are represented by the two variables S_{smt1} and S_{smt2}. The concatenation operation in Line 7 results in the SMT-Lib constraint $(\text{str.++}\ S_{smt1}\ \text{” ”})$. For the following equality comparison, the SMT-Lib constraint is $(not\ (=\ \ S_{smt1}\ (\text{str.++}\ S_{smt1}\ \text{” ”})))$, which is satisfiable and is the currently executed branch in the first run, as the concrete value is assumed to be the empty string in the first execution satisfying this constraint. The example demonstrates how encoding into SMT-Lib using string theory allows a higher abstraction level in the symbolic encoding. In what follows, I will elaborate in further detail on the relation between the SMT-Lib language and $\mathcal{SL}_{\mathcal{J}}$ in more detail, starting with those operations mapping directly from $\mathcal{SL}_{\mathcal{J}}$ to SMT-Lib and continuing with those that do not map directly.

Table 4.2 presents the part of $\mathcal{SL}_{\mathcal{J}}$ that maps directly to SMT-Lib without further modification. The model computed by the constraint solver fits the Java semantics, and no additional error handling is required. The results of these functions overlap in semantic meanings between both languages without any additional result interpretation. Thus e.g., concat(s1, s2) always returns the concatenation of $s1$ and $s2$. Other functions, as IndexOf(s1, s2) do not raise an error if $s2$ is not found in the string but rather return -1 in Java and SMT-Lib semantics. Hence these two semantics fit well.

There are also a lot of operations in $\mathcal{SL}_{\mathcal{J}}$ that do not match the semantic of their

$\mathcal{SL}_{\mathcal{J}}$ Operation	$\mathcal{SL}_{\mathcal{SMT}}$ Operation	Comment
charAt(s1, i1)	(*str.at s1 i1*)	The charAt function requires some error handling in Java not represented in the SMT-Lib function *str.at*.
compareTo(s1, s2)	(*str.* < *s1 s2*) (*str.* <= *s1 s2*)	SMT-Lib has lexicographic ordering operations but they need to be embedded in the evaluation of compareTo splitting the three value result logic to binary decisions.
compareTo-IngoreCase(s1, s2)	–	There is no mapping in SMT-Lib allowing the encoding of the ignore case semantic. Using solver specific operations as toUpper allow to work around this limitation.
equals-IgnoreCase(s1, s2)	–	The same problem as for compareToIgnoreCase applies.

isLetter(c1)
isUpperCase(c1)
	–	It is possible to use *str.to_code* to convert *c*1 into a code point. But afterwards the unicode table defining which code points are within the target domain have to be encoded as well. In practice, we have only achived to encode this for limited ranges on the code point.

join(s1, s2[])	–	There is no way for expressing a join on a symbolic string array yet as we have not really a way to express the capacity of an array symbolically.
lastIndexOf(s1, s2)	(declare-const x Int) (*and* (= (*str.at s1 x*) *s2*) (*not* (*exists* ((*y Int*)) (*and* (< *x y*) (< *y* (*str.len s1*)) (*not* (= (*str.at s1 y*) *s2*))))))	We can encode this using helper variables, but it is leaving the *QF_SLIA* theory as quantifiers are required. Therefore, the encoding is not within the official theory definition of the SMT-Lib anymore as *SLIA* is not defined as official theory.
matches(s1, s2)	(*str.in_re s1* ...)	Depending on the complexity of the pattern involved in *s*2, this is possible. But the pattern contained in *s*2 needs to be transformed to SMT-Lib first.

parseFP(s1, fpSize)
parseInt(s1)
	–	It is not possible to model this in SMT-Lib at the moment (cf. Section 4.4, Linking Numbers and Strings).

split(s1, s2)	–	It is not possible to model a full symbolic split case in SMT-Lib at the moment. Enforcing the equality of the concatenation of the new subparts with the separator s2 and s1 during the execution of one path is possible. (cf. Section 4.4, Encoding Requiring Knowledge About Future Usage.)
reverse(s1)	–	Reversing a string is not supported in todays SMT solvers.
strip(s1)	(*and* (*not* (= (*str.at s1* 0) " ″)) (*not* (= (*str.at s1* (− (*str.len s1*) 1)) " ″)))	While this encoding implies that the first and last character are no whitespaces, it is no possible to express that a string might be shorter after strip in this encoding.

stripIndent(s1)
stripLeading(s1)
stripTrailing(s1)
trim(s1)
	–	See the problem with strip. The same applies for these methods.

toString(fp1)
toString(i1)
toString(c1)
	–	There is no symbolic encoding in SMT-Lib that allows to convert a numeric value in its string representation.

toUpper(s1)
toLower(s1)
	(*str.upper s1*)
	(*str.lower s1*)
		These functions are not supported in the official SMT-Lib standard. CVC4 supports it as a custom interface.

insert(s1, s2, i1)
delectCharAt(s1, i1)
delete(s1, i1, i2)
	-	These function do not have a counter part in SMT-Lib but can be encoded using *substring* to split the existing string and gluing the remaining parts together using *concat*.

Table 4.3: Functions that cannot be mapped from the $\mathcal{SL}_{\mathcal{J}}$ language directly and precisely to $\mathcal{SL}_{\mathcal{SMT}}$ or do not maintain their semantic (adapted from Table 2 in Paper VI [105]).

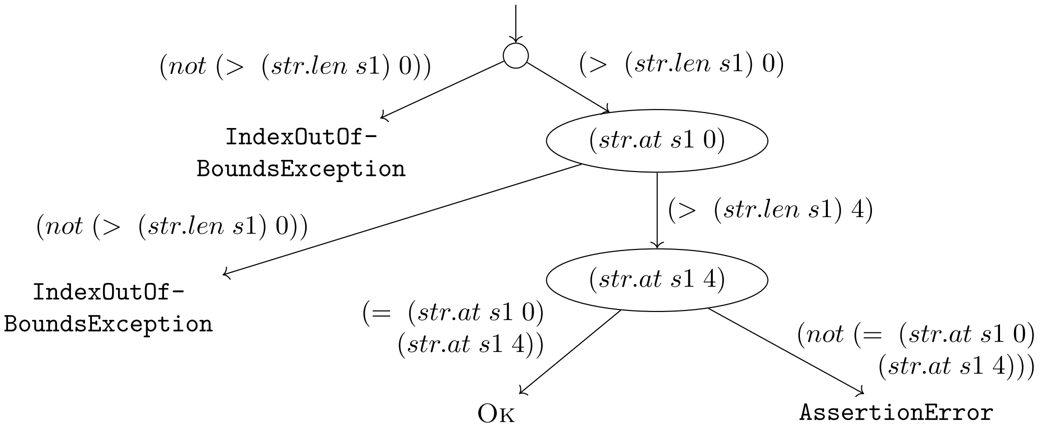


Figure 4.6: The dynamic symbolic execution tree resulting from full exploration of the example in Listing 4.3 assuming that the symbolic string variable is called $s1$. The constraints next to the arrows are the symbolic encodings required to describe the Java semantics for this path. For simplicity, the required bitvector to integer conversion code in SMT-Lib is not shown.

counterpart in SMT-Lib, or have no direct counterparts in SMT-Lib requiring a more elaborate encoding for the Java method using multiple SMT-Lib operations. These functions are listed together with a short explanation in Table 4.3. Following this, I will discuss the encoding challenges that result from integrating error handling, resolving semantic mismatches between Java and SMT-Lib, and encoding regular expressions in more detail. In addition, I will explain how these three problems impactv dynamic symbolic execution on programs with strings.

Faithful Error Handling. Like $\mathcal{SL}_{\mathcal{BV}}$ encoding, $\mathcal{SL}_{\mathcal{SMT}}$ provides a precise model for exceptions and errors in the Java language. As SMT-Lib and Java string semantics vary in error case semantics, a more precise encoding requires a two-step approach like the $\mathcal{SL}_{\mathcal{BV}}$ error modeling presented in Section 4.2. Here, I will describe the approach in more detail for charAt(s1, i1) as an example using the string theory encoding.

Consider the example in Listing 4.3. It creates a StringBuilder object from a nondeterministic string value and assigns it to the variable $buffer$ in Line 3. Line 4 compares the character at position zero and position four. Further, it asserts that these two characters in the nondeterministic string are always equal, which is not the case. As the presented dynamic symbolic execution engines run with the empty string “” value for a new symbolic variable as default in the first execution, the first charAt operation at Line 4 throws

```
public static void main(String[] args) {              1
    StringBuilder buffer =                            2
      new StringBuilder(Verifier.nondetString());     3
    assert buffer.charAt(0) == buffer.charAt(4);      4
  }                                                   5
```

Listing 4.3: The StringBuilderChars03 example from SV-COMP 2022 using charAt on a symbolic string.

an `IndexOutOfBoundsException`. Encoding this path using the *str.at* method in the SMT-Lib is insufficient, as (*str.at* ”” 0) is satisfiable in SMT-Lib theory semantics for the empty string. Instead, $\mathcal{SL}_{\mathcal{SMT}}$ uses a two-layer encoding in such situations allowing for faithful error and exception handling of the Java semantics.

First, it encodes the string length requirements so that the concrete execution does not throw an `IndexOutOfBoundsException` for this specific charAt(...) invocation, as shown at the top of Figure 4.6. Reexecuting the program with some input values that validate the negation of the resulting guard expression enforces the exploration of the other branch in the tree at the root node. On the other hand, if a concrete execution involves a concolic value that runs down the right branch in the decision at this point, the length constraint will check later, during symbolic exploration, whether an exception is possible. Symbolic encoding of the length constraint ensures the creation of a model that triggers this exception if it is used for driving the concrete execution.

Secondly, the operations on the string part are mapped to the *str.at* function in SMT-Lib after the correct string length for the requested indices is ensured. Consequently, four concrete runs are required to explore this small example completely.

Two-step encoding is required because a high-level language such as Java allows two return values in the form of an error or a result value. Thus the SMT-Lib allows the outcome of the operation to be compared against the empty string as a satisfiable model in cases that raise an exception in Java.

Non-binary intermediate decision results. Occasionally, decisions are made in $\mathcal{SL}_{\mathcal{J}}$ that might result in an answer that is not a binary decision. Consider, e.g., thecompareTo(s1, s2) operation in Line 4 of Listing 4.4. It results in an integer value that is either zero, greater than zero, or less than zero. This integer value is interpreted using additional context information to make a binary decision in the *if* conditions (cf. Line 5 and Line 6). For dynamic symbolic execution, the binary decision is the interesting part, as this is the point in time during the execution that adds additional branches to the constraint tree. But it is impossible to capture them while executing compareTo.

```
public static void main(String[] args) {        1
  String s1 = Verifier.nondetString();          2
  String s2 = ”comparisonTest”;                 3
  int res = s1.compareTo(s2);                   4
  if(res > 0){ … }                              5
  else if(res == 0){ assert false; }            6
  else{ … }                                     7
}                                               8
```

Listing 4.4: An example demonstrating the usage of compareTo on a symbolic string.

SMT-Lib supports lexicographic ordering relations semantically, which is equivalent to checking the integer result of compareTo in relation to zero, e.g., the semantic of $compareTo(s1, s2) \ < \ 0$ in the Java language is equivalent to $(< \ s1 \ s2)$ in the SMT-Lib. At the time of encoding, during the invocation of compareTo, it is not known what the future comparison will be. To ensure that all potential effects are triggered, $\mathcal{SL}_{\mathcal{SMT}}$ always encodes all three possible cases as results (greater than zero, equal to zero, and less than zero). Consequently, the symbolic execution engine tries to generate matching inputs, so the compareTo method will be reexecuted up to three times for a complete exploration of the program under analysis. This is a required overapproximation as the analysis might otherwise be incomplete. As the binary decision might split execution into only two paths and merge the other two cases in one execution path (or never perform a decision on the compareTo result), $\mathcal{SL}_{\mathcal{SMT}}$ will potentially add more branches to the symbolic decision tree than are present in the real program. The additional branches are a side effect of describing the potential three-way split in the SMT-Lib semantic. Precise tracking requires knowledge of the future usage of the result, as in the challenge described in Subsection 4.4.

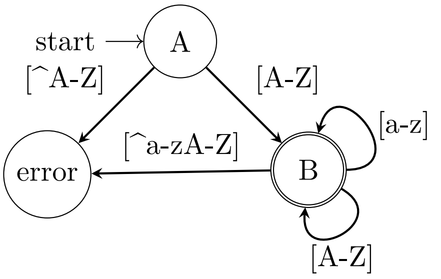


Figure 4.7: Automaton for the Java regular expression "[A-Z][a-zA-Z]*".

Encoding Regular Expressions. As the Java string library offers multiple ways to work with regular expressions, it is not straightforward to encode them. The matches(s1, s2) operation in $\mathcal{SL}_{\mathcal{J}}$ that checks whether a string $s1$ matches the regular expression expressed by $s2$ is the only operation for interacting with a regular expression that can be encoded in $\mathcal{SL}_{\mathcal{SMT}}$, so long as the regular expression does not contain capture groups.

SMT-Lib supports the operation *str.in_re* that is semantically mostly identical to the matches operation in $\mathcal{SL}_{\mathcal{J}}$; the only major difference is how regular expressions are represented. Java expresses them as string, e.g., `s1.matches(``[A-Z][a-zA-Z]*'')`. Figure 4.7 shows an accepting automaton that is equivalent to this regular expression. SMT-Lib encodes the same problem to the following assertion:

$$\begin{aligned}&(\text{declare-const } s1\ \mathit{String})\\ &(\mathit{assert}\ (\mathit{str.in\_re}\ s1\ (\text{re.++}\ (\mathit{re.range}\ "A"\ "Z")\\ &\quad(\mathit{re.}*\ (\text{re.++}\ (\mathit{re.range}\ "a"\ "z")\ (\mathit{re.range}\ "A"\ "Z"))))))\end{aligned} \tag{4.1}$$

The assertion in Equation 4.1 shows that SMT-Lib encodes the paths leading to accepting in the automaton of Figure 4.7 resulting from parsing the Java regular expression into its tokens. Hence $\mathcal{SL}_{\mathcal{SMT}}$ requires a parsing frontend that maps the Java regular expression string onto the regular expression automaton and encodes the automaton in SMT-Lib. Further, the frontend has to replace Java specific character classes, e.g., \p{Upper}, with more general representations of the same semantic in SMT-Lib: e.g., $[A - Z]$. $\mathcal{SL}_{\mathcal{SMT}}$ replaces the character classes and uses in continuation the brics automaton library[4], part of the JSA project [41], to lift the string representation to an automaton. It encodes the automaton representation in SMT-Lib afterward. This way, the matches operation from $\mathcal{SL}_{\mathcal{J}}$ is often encodable in $\mathcal{SL}_{\mathcal{SMT}}$, so long as it does not use capture groups, as they cannot be represented in SMT-Lib.

Brief Discussion. All in all, of the 72 methods ticked in $\mathcal{SL}_{\mathcal{J}}$ in Table 4.1, $\mathcal{SL}_{\mathcal{SMT}}$ encodes 30 methods completely using the 13 symbolic string theory operations shown in

[4] `https://www.brics.dk/automaton/` {last accessed: February 2022}

Table 4.2, and 11 partially using either prototypes or requiring solver-specific features, e.g., toLower in CVC4. Table 4.3 discusses the limitations of the only partially encoded operations. Operations with regular expressions without capture groups such as the matches method on *String* are supported in $\mathcal{SL}_{\mathcal{SMT}}$. However, the support of methods defined on the *Character* class is limited in $\mathcal{SL}_{\mathcal{SMT}}$, leading to 14 unsupported methods in $\mathcal{SL}_{\mathcal{SMT}}$. The remaining 16 methods in $\mathcal{SL}_{\mathcal{J}}$ that are not supported in $\mathcal{SL}_{\mathcal{SMT}}$ are mostly related to the number and string conversion, trimming of strings, and splitting or joining strings.

4.4 Comparison and Open Challenges

$\mathcal{SL}_{\mathcal{J}}$ defines a subset of the Java standard library that allows dynamic symbolic execution of Java programs. With $\mathcal{SL}_{\mathcal{BV}}$ and $\mathcal{SL}_{\mathcal{SMT}}$ I have presented two strategies for encoding parts of $\mathcal{SL}_{\mathcal{J}}$, answering SRQ2. The $\mathcal{SL}_{\mathcal{SMT}}$ is more promising in comparison with $\mathcal{SL}_{\mathcal{BV}}$ and supports a larger part of $\mathcal{SL}_{\mathcal{J}}$ than $\mathcal{SL}_{\mathcal{BV}}$. Especially the encoding of more complex string operations such as matches(s1, s2) is easier in today's string theory integrated into SMT-Lib. To conclude the chapter, I will discuss three limitations to the presented approaches answering SRQ3. The limitations are related to splitting or joining strings, regular expressions with capture groups, and the de- and serialization of values between strings and numbers.

Encodings Requiring Knowledge About Future Usage. A few methods in the Java string library are hard to encode during dynamic symbolic execution, as they break pure string theory and turn a string value into an array or vice versa. One of these methods is split(s1,s2), which splits string $s1$ on every match of $s2$. The problem is that the result is an array of strings that are subparts of the original symbolic string. Depending on later usage of the array, the original string will require a certain shape. The constraints need to reflect this shape so that the resulting model can be used for driving dynamic execution along the desired paths. I will explain this in more detail with an example.

```
public static void main(String[] args) {                    1
  String sentence = Verifier.nondetString();                2
  String[] tokens = sentence.split(” ”);                    3
  int i = 0;                                                4
  for (String token : tokens) {                             5
    if (i == 3) assert token.equals(”genneration”);         6
    ++i;                                                    7
  }                                                         8
}                                                           9
```

Listing 4.5: The TokenTest02 example from the SV-COMP Java track demonstrating a split on a symbolic string.

Consider the example in Listing 4.5. Line 2 generates a symbolic string without any further constraints. This string is split in Line 3 into a `tokens` array on any occurrence of whitespace. In Line 5, the resulting string array is iterated. If the array has that many entries, Line 6 compares the fourth element of the array in an assertion condition with the word “genneration” for equality. These are the relevant parts of the example to demonstrate the problem in symbolically encoding the effect of the split operation.

As the for-loop in Line 5 iterates the array resulting from the SPLIT method, the number of whitespaces enforced in the symbolic string influences how often the loop is unrolled. The information about the loop unrolling and the number of desired iterations is only available after execution of the SPLIT method. In the example, it is impossible to encode the constraints on the string value that forces the SPLIT operation of the concrete string to produce an array with at least four elements during execution. This is because, at least in standard symbolic forward execution, information on a desired array size of greater or equal to four is unavailable during the execution of SPLIT. Instead, the result value needs an additional annotation that allows encoding of the string after executing the branch. In the concrete example, the analysis needs a way to transfer the knowledge that the layout influences string array size to the symbolic string. I call this the "encoding knowledge about future usage" problem; but encoding these constraints at the end of the path is only a partial solution of the problem. The real problem is that no interpretation of a function that links the array of a split result with the original string in SMT-Lib is yet available. Therefore, the effects of the SPLIT method are not expressible in SMT-Lib.

The JDART prototype encodes the string after the path, making the array one element bigger with each iteration. This happens as a side effect of the for-loop that tries to increment the array size by one after each iteration. Therefore, this procedure is not unsound; but the prototype raises another problem: the split of the symbolic string is unbounded. The resulting token array can be arbitrarily large in theory and will never terminate, following the JAVA language definition requiring many (potentially arbitrary) concolic executions of the program. However, considering the bound on string lengths used in the JVM standard implementation, it will still span an impressive domain space for the array values, hindering the termination of the symbolic search. The symbolic reasoning component requires knowledge of the source code structure in order to evaluate whether all reachable branches worth exploring have been analyzed. Unfortunately, in its current form, this information is not available for the forward symbolic execution presented here.

Similar considerations as for the SPLIT(S1, S2) method apply for encoding JOIN(S1, S2[]) and REPEAT(S1, I1). The JOIN method is the semantic reverse operation of the SPLIT method. All challenges for encoding SPLIT apply equally for JOIN. REPEAT(S1,I1) can be expressed as a JOIN operation with an array of size i1 that always has the same content. The glue string is the empty string. Therefore, REPEAT inherits the same problems as encoding JOIN.

Regular Expressions with Capture Groups. As described above, it is possible to encode simple checks against regular expressions without capture groups using the *str.in_re* operation of SMT-Lib. Using capture groups makes it impossible as it is hard to model the general behavior of capture groups in SMT-Lib at the moment. I will explain the problem using the example in Listing 4.6.

The example demonstrates the creation of a pattern in the JAVA regex language in Line 2. The pattern requires a string to have a 'W' followed by a sequence of any

```
public static void main(String[] args) {                                          1
  Pattern expression = Pattern.compile("W.*\\d[0-35-9]-\\d\\d-\\d\\d");           2
                                                                                  3
  String string1 = Verifier.nondetString();                                       4
                                                                                  5
  Matcher matcher = expression.matcher(string1);                                  6
                                                                                  7
  while (matcher.find()) {                                                        8
    System.out.println(matcher.group());                                          9
    String tmp = matcher.group();                                                 10
    assert tmp.equals("WWWWW's Birthday is 12-17-77");                            11
  }                                                                               12
}                                                                                 13
```

Listing 4.6: The RegexMatches02 example from the SV-COMP Java track demonstrating the usage of a *Matcher* and a *Pattern* object on a symbolic string.

characters. A series of three pairs of digits separated by a dash ends the pattern. The first pair requires a digit other than four as the second member in the pair.

The pattern is matched against a symbolic string resulting in a *Matcher* object in Line 6. A *Matcher* object is bound against a specific pattern instance and allows occurrences of this pattern to be found in a string. Each occurrence can be extracted as a group and eventually subgroups if the pattern contains them. The example searches for the pattern using the FIND() method on the *Matcher* object in Line 8 and extracts each occurrence of a match using the GROUP method in Line 10.

Encoding the effects of the *Matcher* object symbolically is challenging, as the effects depend on the internal state, which is influenced by previous operations executed on the object. The encoding, therefore, has to track the internal state of the *Matcher* instance, resulting in different return values of the method invocations on the instance. If a call to the FIND() operation was successful, a GROUP() must return a string that is the content of the last find result. Otherwise, the string should be empty. If a non-empty string is returned, it inherits restrictions on the content applied by the pattern. Other operations, like calling REPLACE(...) after FIND(), alter the state of the string analyzed by the *Matcher*.

The while-loop in Line 8 depends on the FIND() outcome. As a symbolic string is not further bounded, a FIND() will in theory always find the next match. Therefore, in theory, the while-loop is unbound and iterates forever. In practice, there is an upper bound in most concrete implementations of string in the Java standard library. Nevertheless, the resulting search space is enormous. Considering the internal state of the *Matcher* object during the execution of a path, it is possible to drive down the execution along different paths in a constraint tree using already existing operations. But as the search space is not bounded, dynamic symbolic execution is a pure search method at this point that will in theory never terminate its search without further bounding.

In addition, the Java pattern language allows definition of capture groups within a pattern, e.g., “A.*(Cat)C”. Hence, there are different capturing groups (e.g., the string “Cat”) that might be extracted as a subgroup from the main match group (e.g., A.*CatC).

However, as SMT-Lib does not support any of these operations, it is only possible to symbolically encode the structure of a string that drives down the concrete execution on a certain path if the structure of that path is known. This is similar to encoding the split operation.

The problem of combining the regular expression language with dynamic symbolic execution has been discussed by Loring et al. [88] in the context of analyzing JavaScript. They propose a CEGAR-based approach to create models suitable for driving the concrete execution during dynamic symbolic execution of JavaScript programs. To the best of my knowledge, their approach has not yet been adapted to the JVM.

Linking Numbers and Strings. Especially in Web Applications, numbers are sometimes serialized into a string-based format like JSON and are later deserialized back to numbers from the string. While the value exists for parts of its life span as a string and other parts as a number, it is still the same value in the semantics of the program. Analyzing these values requires linking the string representation with the number representation in the symbolic encoding. This is not supported in SMT-Lib yet. We have experimented with a preliminary approach in JDart for establishing this link in the analysis engine. Given the lessons learned, this approach is not ported to GDart as it is not precise. I will explain the problem in more detail below and describe our preliminary solution proposal.

```
String stringNumber = readerBuffered.readLine();                    1
if (stringNumber != null) {                                         2
  try {                                                             3
    data = Float.parseFloat(stringNumber.trim());                   4
  } catch (NumberFormatException exceptNumberFormat) {              5
    IO.writeLine("Number format exception");                        6
  }}                                                                7
```

Listing 4.7: Example of deserialization from String to Float.

JDart uses a preliminary approach for connecting the string and a numeric theory for floats, doubles, and integers on the Java data type level. Listing 4.7 shows a snippet taken from the Juliet benchmark [29] suite for comparing the performance of static analyzers for Java. Assume that the *readerBuffered* object has been set up to read from a nondeterministic stream not shown in the preceding code. Therefore, all lines read by the readLine() method invoked on the *readerBuffered* object will be nondeterministic string values with symbolic variables for the string. Consequently, *stringNumber* in Line 1 is a symbolic String. Next, this string is trimmed so that all whitespaces are removed and the value is parsed into a float value in Line 4. There are two possible outcomes for the parsing method: the string is parsed successfully into a float value, or a number format exception is raised. This decision has been modeled using a Boolean value in the preliminary prototype. If the Boolean value indicates an error has been raised, we raise an error in the concrete behavior. Otherwise, a symbolic float value is created that represents the number originating from parsing this string (without actually parsing it). We add a symbolic annotation to the newly created float to indicate that it is parsed from a string. The preliminary prototype allows this float value to be used in branch conditions and other value checks. Further, the symbolic encoding allows generation of the matching floats to trigger different branches if the parsed float value is part of the branching condition. However, the float is treated like a normal float variable and not parsed from the string. The concrete values for the float are handed over to JDART as for any other float value.

The downside of this approach is that the symbolic string value and the symbolic float value from this string are not linked. The string value cannot be parsed into the float value outside of JDART during concrete execution and therefore undermines the goal of JDART. The prototype shows the potential inherent in this line of work for linking the different theories in the context of dynamic symbolic execution. Nevertheless, some post-processing is required to turn it into a fully sound approach that works independently of JDART and generates string inputs that are parsed into the right float values on any JVM. In its current state, this is possible as a post-processing step of the model before executing a single concrete path, allowing the generation of a concrete string from the computed concrete float value.

A rock-solid solution must first check the string's integrity before parsing and take into account string operations that might restrict the domain range of the parsed number. For example, assume a program asserts on a string s_n that it does not contain the digit '4'. Assume further that s_n encodes a number in decimal encoding. Calling PARSEINT(S_n) after the assert, it is impossible for the parsed integer to compare equally to an integer number containing the digit '4'. Unfortunately, JDART's preliminary prototype does not incorporate these checks and, therefore, we have not ported it to GDART. At the same time, the example described here is the reason why linking the value of two variables in these two domains is in the long term desirable for the analysis of JAVA.

5 Jaint

The JAINT framework combines dynamic symbolic execution and dynamic multi-color taint analysis for JAVA web applications in a single tool. It is the core contribution of this thesis, and its main ideas are published in Paper I [107]. This chapter first describes in Section 5.1 the idea of multi-color taint analysis. Then it explains why taint analysis is suitable for detecting security weaknesses. Section 5.2 describes how dynamic symbolic execution is used as a path enumerator ensuring multi-color taint analysis executes all reachable branches within the search space of the symbolic search. This loose coupling forms the core combination of the two techniques. Following the principle of separation of concerns, the two techniques are implemented separately into the same JVM. The accuracy and completeness of the analysis result from the implicit interaction of the two techniques, but they are coupled loosely, exchanging only a few pieces of information. JAINT is configurable for different security weaknesses as described in Section 5.3. The chapter concludes with a technical discussion of the steps involved in implementing multi-color taint analysis in JPF-VM as part of JAINT and how the architecture supports the configurability of JAINT.

5.1 Dynamic Multi-Color Taint Analysis

As observed in the Introduction, many different applications of taint analysis are described in the literature, from stack protection in assembly based languages (e.g. [45, 67, 135]), through preventing injection attack in web applications (e.g [72, 86, 109, 111]), to tracking information flow(e.g. [5, 48, 64, 74, 124]). The target of the JAINT framework is security analysis for typical injection attacks on JAVA web applications. Dynamic and explicit taint tracking is a good fit for injection attacks. Implicit information flow across control-flow instructions is less relevant for injection weaknesses, because injection requires manipulation of a string that transforms into a command-line or SQL statement. For it to be possible to negatively attack such a statement using implicit flow, the program must already encode the malicious parts. In the context of analyzing JAVA web applications, it is less likely that these security weaknesses be introduced into the code base without malicious intent. For example, source code that constructs different SQL statements based on parameter values in the condition without any explicit flow and still allows information leakages must have a very strange design. I expect such code to be detected during conventional software review, as this not only requires judgment on security impact but also on the general design of a component under review.

From a theoretical viewpoint, the explicit spreading of taint in a program is a flow problem. Running a security analysis is semantically the same as ensuring that the resulting taint flow across all data assignments in a program from an untrusted *source* to a protected *sink* is zero. As this data flow is sometimes desired in the design of an application, there are guard functions that check the data from the untrusted source and turn it into trusted data after inspection. Taint analysis calls this process of inspecting 'data sanitization'.

The CWE database defines multiple security weaknesses. Each of these security weaknesses defines different protected sinks and eventually different untrusted sources. In the case of SQL injection, CWE 89, and Cross-site Scripting (XSS), CWE 79, both weaknesses share input from an HTTP request in a Java servlet as source, but the SQL injection pattern protects database statements as sink, and the XSS pattern protects the HTTP response object as a sink. Therefore, we have one flow problem per security weakness in the analysis. Some taint based security analyses analyze source code for the presence of multiple security weaknesses simultaneously. Technically, they run multiple taint analyses in parallel. To distinguish them, each analysis tracks a flow with a different name. In the taint analysis slang, these names are called taint color. . In what follows, I will demonstrate the meaning of multi-color taint analysis by example, discuss different sanitization approaches, position Jaint against them, and summarize the characteristics of multi-color analysis.

Multi-Color Taint Analysis by Example. Consider the example of an SQL injection in Listing 5.1. The concrete SQL injection weakness in the example occurs if some data read from the request object passed to the `doPost`-method in Line 1 reaches the arguments of the `prepareCall` in Line 15 without prior sanitization. If such data flow exists, it allows a potential attacker to influence operations executed on the database. SQL injection is a common security weakness often occurring in software that writes SQL queries to interact with databases on their own or by misusing an SQL abstraction layer, e.g., an object relation manager such as Hibernate[1]. The Common Weakness Enumeration (CWE) lists this security weakness as CEW-89[2].

Oversimplifying the internals of the `getHeader` function, which is called twice and requires string comparison, the method has two possible execution paths before reaching the `prepareCall` that is part of the statement beginning in Line 15 as shown in Figure 5.1. The right hand side of the figure shows the execution path for the case where the request object contains a header with the key "BenchmarkTest00008". The code adds some data from the header value to the *param* variable, and this data is passed to the arguments of `prepareCall`. As this forms a data flow from an untrusted source to a trusted sink vulnerable to the SQL injection weakness, it violates the taint policy and detects the weakness in the source code. The left side of the figures shows the execution path in the example where no data from the request object reaches the arguments of `prepareCall`.

To make the flow of data visible, explicit dynamic taint analysis marks all data originating from a taint source with a taint mark. For the analysis presented in this paper, the taint marks have a color. These colors are added to values on the heap and primitive values on the stack as additional annotations similar to the symbolic annotations used

[1] `https://hibernate.org`{last accessed: February 2022}

[2] `https://cwe.mitre.org/data/definitions/89.html`{last accessed: February 2022}

```
public void doPost(HttpServletRequest request,                    1
    HttpServletResponse response)                                 2
    throws ServletException, IOException {                        3
    // some code                                                  4
    String param = ””;                                            5
    if (request.getHeader(”BenchmarkTest00008”) != null) {        6
        param = request.getHeader(”BenchmarkTest00008”);          7
    }                                                             8
    param = java.net.URLDecoder.decode(param, ”UTF−8”);           9
    String sql = ”{call ” + param + ”}”;                          10
    try {                                                         11
        java.sql.Connection connection =                          12
            DatabaseHelper.getSqlConnection();                    13
        java.sql.CallableStatement statement =                    14
            connection.prepareCall( sql );                        15
        java.sql.ResultSet rs = statement.executeQuery();         16
        DatabaseHelper.printResults(rs, sql, response);           17
    } catch (java.sql.SQLException e) {                           18
        if (DatabaseHelper.hideSQLErrors) {                       19
            response.getWriter()                                  20
            .println(”Error processing request.”);                21
            return;                                               22
        }                                                         23
        else throw new ServletException(e);                       24
    }                                                             25
}                                                                 26
```

Listing 5.1: The testcase 8 demonstrating an SQL injection vulnerability taken from the OWASP Benchmark.

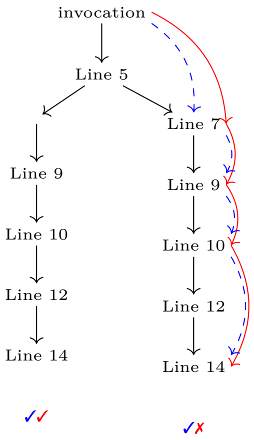


Figure 5.1: The graph shows a simplification of different paths across the example in Listing 5.1. The continous red arrows show how the *sqli* taint information travels across the execution from the request object to the `prepareCall` invocation. The dashed blue arrows are the *cmdi* taint information.

in dynamic symbolic execution. The taint colors associated with a value are always connected with the value inside the JVM running the analysis. The taint mark in this example is colored red representing the *sqli* tag, as it tracks information specifically for SQL injection. The taint makrs are assigned to all data originating from the request object before calling the doPost method. Taint analysis requires a similar method driver for the concrete execution as dynamic symbolic execution. For Jaint, the driver injects the taint marks before invoking doPost.

After the taint marks are attached to the values, they travel across the system with the value and are propagated during primitive or sometimes more complex operations. Figure 5.1 shows how the taint marks are tracked across the lines for the two main branches in Listing 5.1 resulting from the branching statement in Line 6 of the example. In Line 7 of the example, a value read from the request is assigned to the variable *param*. If this line is in the path, the *sqli* taint mark associated with the request object transfers by the propagation rule to the method result and with the return value to the *param* variable. In the execution that propagates the *sqli* taint mark in Line 7, the taint mark is later propagated through the method call in Line 9 and the string concatenation in Line 10. The taint analysis instruments and intercepts the API level methods for string operations so that the propagation semantic can be injected into the execution in the same way as described for symbolic encodings in Section 3.2. For propagation in the string concatenation in Line 10, the append method of the *StringBuilder* class is replaced with a substitution method that propagates the taint from the parameter to the method result. As a result, the *sql* variable receives the taint mark. The statement that begins in Line 12 does not influence the taint propagation. The statement starting in Line 15 prepares a new callable statement using the *sql* parameter. SQL injection policy describes this as a security weakness if the *sql* parameter contains untrusted data. As this variable carries the taint, this violates SQL injection policy and detects a weakness in implementation. The red ✗ in Figure 5.1 shows that on this path, the SQL injection policy is violated, and the red arrows show the described taint propagation flow between the statements. The other branch in Line 6 of the example does not execute the call that propagates the taint mark from the request object to the *param* variable. Hence this path does not violate the policy. The ✓ signifies that the policy holds on this path.

There are also, however, other types of injection attacks, e.g., cross-site scripting attacks (XSS) described in CWE-79[3]. The sources for XSS attacks are the same as for SQL injection, but the vulnerable sinks are different. In the example, the blue taint color marks the traveling of the *xssi* taint mark. The taint engine propagates it in the same way as the red *sqli* mark, but there is no sink corresponding to the blue analysis. We can see how XSS taint policy holds on both paths, and no XSS weakness is reported. The simultaneous check for both taint policies makes it a multi-color taint analysis. It only propagates the flow along with a data value, so it is an explicit flow analysis.

Sanitization. So far, the example only shows taint propagation. Some functions inspect data or compute constants, which therefore stop propagation and cut taint flow. Functions that cut taint flow after inspection are called explicit sanitization functions. The other effect—operations that compute constant values and implicitly cut taint flow—are called implicit sanitization functions.

Explicit sanitization functions are defined per analysis. A method that ensures escaping for an SQL statement does not necessarily do so in a string that allows, e.g., XSS attacks. Hence explicit sanitization functions are part of the taint definition for a specific taint color. The Jaint framework explicitly supports definition of sanitization methods as part of the specification language. Taint marks are cleared on return from sanitization: Jaint intercepts the return statement and cleans the specific color from the result.

Implicit sanitization is often less dependent on the specific taint color and property associated with certain operations. A standard example, also raised as a question in reviews of Paper I [107], is multiplication with zero: $B = A * 0$. Multiplying a value with zero always returns zero as a product irrespective of the other factors involved in the multiplication. As a side effect, this multiplication destroys all taint information. Implicit sanitization might be implemented in the runtime as well. But it is not always trivial to name cases leading to a constant result that cancels the taint flow for a specific

[3] `https://cwe.mitre.org/data/definitions/79.html` {last accessed: February 2022}

function. As implicit sanitization adds complexity to the execution of a byte code, it impairs execution time by slowing down the JVM. Moreover, a calculation always leading to a constant value can also be replaced using a constant assignment.

From my point of view, constant operations are a code smell that should be removed from the code base, and it is not worth complicating dynamic taint instrumentation to avoid them. Rather, it is worth investing in a bytecode analyzer that detects constant operations and allows optimization of programs by removing these bytecodes. For this reason, Jaint does not support implicit sanitization in its current implementation. This negatively influences Jaint's pprecision, as it in theory allows false positives. The decision not to implement implicit sanitization is the only thing limiting Jaint's precision in answering question DQ3 compared with the ideal theoretical result. The architecture of the taint tracker as presented for Jaint allows checks to be implemented if they are legitimately part of the code. All it tracks is extending the bytecodes with these checks.

To assess the impact of not implementing implicit sanitization in Jaint by default, I implemented a small bytecode analyzer on top of the ASM library [34] checking for multiplications with zero and xor operations that lead to a constant result. To my surprise, I found a single line in Jenkins core that led to multiplication with zero: *return 31 * (channel != null ? channel.hashCode() : 0) + remote.hashCode();*. I think it is worth discussing whether the line should be rewritten for this case. The multiplication can be moved inside the ternary operator, making constant multiplication with zero unnecessary. All in all, implicit sanitization is not a roadblock to analyzing Jenkins core given the assessment result. Splitting implicit sanitization analysis from dynamic taint analysis promises better performance in general, as the bytecodes do not have to execute additional checks on every execution. I agree that constant operations involving implicit sanitization are more important in code bases that need to defend against timing-based side-channel attacks. However, implicit sanitization can become a roadblock for analyzing such systems if constant operations are intensively used to mask different path timing.

Characteristics of the Analysis. The proposed taint analysis is dynamic, implying that it checks only the currently executed path. Further, it only tracks explicit taint flow, limiting applicability to analyses that reach a verdict by explicit flow alone. For injection attacks, this limitation is reasonable, as explained above. As presented in Paper I [107], the proposed analysis instruments runtime and therefore does not require alteration of the source code under analysis: sanitization is fully supported for explicit sanitization functions specified in the taint policy. Detected policy violations are precise except in those cases where implicit sanitization secures the system. As the analysis applies only to the currently executed path, it cannot on its own judge the presence of security weaknesses in a complete program.

5.2 DSE as Path Enumerator: From Fuzzing to Verification

The previous section has explained how the basic concept of multi-color taint analysis works. The presentation of its characteristics closed with the observation that taint only checks the branch concretely being executed. However, precisely along this branch it will find security weaknesses with the explained limitation on implicit sanitization. What is missing from here to scaling multi-color taint analysis into an automated security analysis for a complete program is an automated generator of all reachable execution paths that require checking. The analysis must run all of these concretely at least once.

Combining DSE with Multi-Color Taint Analysis. Enumerating all reachable paths is precisely the strength of the dynamic symbolic execution component. Dynamic symbolic execution enumerates paths and runs them with the multi-color taint analysis enabled in the concolic executor. This describes the core of the combination between dynamic symbolic execution and multi-color taint analysis. If the taint analysis detects a property violation, it informs the dynamic symbolic execution engine and, in the case of Jaint, this stops the analysis with a property violation. Nevertheless, apart from this slim communication interface, both are implemented as independent modules run within the same JVM. This is the fundamental architecture for combining dynamic symbolic execution with multi-color taint analysis. Accordingly, it answers DQ2.

Recall now the example of the doPost method in Listing 5.1 and the analysis graph in Figure 5.1. With the default values used for symbolic strings, the symbolic value representing the header object is an empty string in the first execution of the program. The left hand branch in Figure 5.1 is executed first. After this concolic run, there is a constraint in the tree that checks whether the symbolic header value equals "BenchmarkTest00008". The branch satisfying this constraint generates input driving the analysis down the right hand path of Figure 5.1 in the concrete run. This time, the taint analysis finds a property violation.

We decided in the context of Jaint to implement both taint analysis and dynamic symbolic execution within the same JVM. The theoretical coupling also works if dynamic symbolic execution generates the inputs driving the concrete run down the different paths and generates them in an intermediate exchange format. Then, in a second step, a driver for the taint analysis might use these input values for analyzing the program. Suppose dynamic taint analysis and dynamic symbolic execution are run in two separate steps. In that case, there is no benefit in using dynamic symbolic execution rather than symbolic execution as a path generator for the taint analysis. Symbolic execution has the same required information for computing inputs for running every path in the constraint tree it produces. Synergy effects emerge from running both of them in parallel within the same JVM. Found security weaknesses can be used to limit the search space, speeding up the analysis. If the analysis is only interested in finding one or no security weakness, terminating the analysis on the first found weakness will save resources compared to running the full dynamic symbolic execution upfront. The two analyses share the complete infrastructure for executing their concrete runs.

Random Drivers. In theory, any random input generation is suitable to trigger reruns of the different execution paths. Over time, the branch coverage of the program under analysis should increase with random inputs. This is the general idea, e.g., implemented in Randoop [116]. However, random path enumeration does not guarantee completeness of the covered state space. So combining Randoop with multi-color taint analysis is a pure testing approach that works as a security fuzzer for JAVA programs. Dimjašević et al. [58] presented JDOOP, a tool combining JDART and RANDOOP. They have shown that JDOOP covers more branches within the same period than RANDOOP alone on the SF110 benchmark. I have not run controlled experiments to compare the achieved coverage of JDOOP with today's version of JDART. However, this experiment does show that investing in a more structured approach than random enumeration for enumerating the search space can save resources. So JDOOP (instead of RANDOOP) is potentially a better security fuzzer. As structured enumeration prevents repeated reexecution of the same branch, it manages the available resources better than random execution. In addition, structured enumeration gives stronger guarantees.

Guarantees Resulting from Applying DSE as Driver. These stronger guarantees affect the soundness and recall of the analysis. Dynamic symbolic execution systematically enumerates the reachable search paths and ensures the soundness of the combined analysis for the test space covered by the dynamic driver so long as the enumeration ends. How the driver used for dynamic symbolic execution slices the program, therefore, significantly influences the recall performance of JAINT. If the driver defines a symbolic search space that covers the interesting properties, and the SMT backend solves all resulting SMT constraints, JAINT will find the security weakness with perfect recall results. But if the security weakness is outside the enumerated state space, JAINT will not find the weakness, as the concrete driver makes the required path unreachable within the nondeterministic part of the analysis. If the nondeterministic state space is infinite or the SMT solver is unable to solve the constraints involved, so that symbolic execution does not terminate, JAINT cannot guarantee to find a weakness in the program. Whenever the search does not terminate, it harms the recall potential of JAINT. These considerations together answer DQ3.

From Testing to Verification. The structured enumeration performed by dynamic symbolic execution ensures soundness—if the enumeration ends—making JAINT a verification framework. A random driver in JAINT is a fuzzer only allowing searches for weaknesses. Using structured enumeration instead of random enumeration, searching for security weaknesses along these paths becomes a verification approach with the possibility of proving the absence of weaknesses. While JAINT carries out the analysis, the intermediate results can be documented and used as an intelligent white-box fuzzer. . If JAINT analyzes the program but the search is not terminated, it will give the same guarantees as a fuzzer. If the analysis terminates, it will give the same guarantees as a software verifier. This way, the JAINT framework provides value as a software verifier in many cases and can still be used for intelligent security testing in the remaining cases.

Regarding the time of detection vs. time of attack discussion mentioned in the Introduction, it is true that, in order to observe them, JAINT requires an execution environment that executes attacks dynamically. Given the framework's capabilities, it is possible to run them within a sandbox during a security test phase. Consequently, production data is not compromised, as the security test is carried out in an automated fashion upfront. Given that any weaknesses found will be removed from the program, this efficiently prevents attacks during production. The stronger guarantees of JAINT in comparison with fuzzing allow the security test to be run in the CI pipeline before deployment, reaching a final verdict within a predefined time frame and thus making the discussion regarding time of detection and time of attack dispensable. In theory, the proposed framework detects a security weakness in implementation before it becomes exploitable in a deployed production setting. Ideally, it will prove the absence of security weaknesses before deploying the program.

5.3 Jaint's Configuration Language

A core problem for defining taint analysis is specifying where taint is injected, removed, and checked in the program's execution. A single analysis consists of a definition for taint injection places (the taint *sources*), a definition of the vulnerable targets (the taint *sinks*), and eventually the set of sanitization methods. As Jaint supports multi-color taint analysis, the taint color definition completes the quadruple. The language, examples, and how it is used to analyze the OWASP benchmark are explained in the second half of Paper I [107]. I will not, then, show the details of the language itself here but give an example of the SQL injection taint specification used in Paper I and compare it with the configuration approach of FireSecBug.

$$\begin{aligned} \mathit{Src} &::= \mathit{sqli} \leftarrow (\_ : *\texttt{HttpServletRequest}).\texttt{get}*() \\ \mathit{Sink} &::= \mathit{sqli} \rightarrow (\_ : \texttt{java.sql.Statement}).*(\texttt{sql}), \\ &\qquad\quad (\_ : \texttt{java.sql.Connection}).*(\texttt{sql}), \\ &\qquad\quad (\_ : \texttt{org.springframework.jdbc.core.JdbcTemplate}).*(\texttt{sql}) \end{aligned} \tag{5.1}$$

Equation 5.1 defines the taint analysis required for analyzing the SQL injection in the OWASP Benchmark. We see the name of the taint color *sqli*, the HttpServletRequest object as the source of the taint flow, and a collection of methods used for database access in Java as protected sources. The configuration language determines that the parameter with the name "sql" must not be tainted with the *sqli* taint color invoking any method on these classes.

Most tainting frameworks require defining the taint analysis one way or the other. Sometimes the definition is textual, sometimes hard coded in the source code. For Jaint, we decided to use a textual format and tried to make it easily understandable. We hope that Jaint's configuration language makes it easier to express analysis even for chief security officers who are not always Java cracks. We have not evaluated this empirically as part of Paper I [107], and I have not done it for this thesis either. Nevertheless, let me give an example for the configuration of the taint sinks in FindSecBug first, before explaining why I think Jaint's configuration language is easier.

An existing example for configurable static analysis is the FindSecBug[4] tool with its configuration language. Specifying the data types requires a lot of internal knowledge about encoding data types in JVM and explicitly listing every variant of a method call.

[4] `https://find-sec-bugs.github.io` {last accessed: February 2022}

The vulnerable sink *prepareCall* for SQL injection used in Listing 5.1 looks as follows for FindSecBug:

$$java/sql/Connection.prepareCall(Ljava/lang/String;)Ljava/sql/CallableStatement; : 0$$
$$java/sql/Connection.prepareCall(Ljava/lang/String;II)Ljava/sql/CallableStatement; : 2$$
$$java/sql/Connection.prepareCall(Ljava/lang/String;III)Ljava/sql/CallableStatement; : 3$$

In comparison, JAINT's configuration language expresses that no argument called *sql* defined in a method declaration part of the *Connection* class must be tainted. Only the following part of the overall sink definition covers the same ground as the FindSecBug definition:

sqli → (_ : java.sql.Connection).∗(sql).

Comparing FindSecBug's language with JAINT's language, JAINT's language is easier, as it does not even require the method to be explicitly named. The configuration language allows taint to be forbidden on the basis of API's parameter name, which is consistent *sql* for the query string in the *java.sql.Connection* class. All information required to define this sink is available in the JAVA API specification (sometimes called JAVA Docs). FindSecBug requires the user to list explicitly each method that is a sink, using the JVM specific type definition. This requires deeper knowledge of the internals of type name representation inside JVM. Further, all overloaded sub-methods have to be specified. Whenever the signature of a method is changed, the FindSecBug configuration needs to be adapted. JAINT's configuration language allows more flexibility and defines the taint analysis in a policy style, abstracting more from the concrete implementation than does FindSecBug.

Program Query Language (PQL) [92] is the only other configuration language I am aware of for taint analysis in JAVA that is directly used to configure the analysis. PQL is used in the tool SECURIFLY, which is part of the same paper for taint-based security analysis. But the main focus there is on combining static and dynamic analyses. The language is more advanced and complex than in JAINT's configuration language, as it targets more complex use cases such as injecting errors during testing. Comparing the two languages: just from reading the paper I have not been able to give an example in PQL that reassembles the sink protection rule for the same SQL sink described above for JAINT and FindSecBug.

5.4 Taint Propagation in Jaint: Taint and Value Monitors

Making the analysis flexible and completely configurable is the main goal of the design of JAINT and, therefore, part of DQ1. For the final design of JAINT, the goal is a complete configuration of the analysis using the configuration language without manually

```
public class TaintContainer {
  public Boolean isCrossSiteScriptingTainted = false;
  public Boolean isSQLInjectionTainted = false;
  ...
  public TaintContainer combine(TaintContainer taintContainer) {
    if (taintContainer == null) {
      return this;
    }
    TaintContainer container = new TaintContainer();
    container.isCrossSiteScriptingTainted =
      this.isCrossSiteScriptingTainted   taintContainer.isCrossSiteScriptingTainted;
    container.isSQLInjectionTainted =
      this.isSQLInjectionTainted   taintContainer.isSQLInjectionTainted;
    return container;
  }
}
```

Listing 5.2: Examples taken from the *TaintContainer* used for analyzing the OWASP Benchmark in the reproduction package [106] of Paper I [107].

implementing any parts of the taint propagation. This section will explain how the design uses so-called taint containers that serve as a wrapper class for the concrete taint analysis, so that the design supports the required flexibility for reaching the design goal. The taint containers serve as a wrapper class for the concrete taint analysis. This section focuses on the technical details of implementing the multi-color taint analysis in Jaint.

It is possible to compile the dynamic taint engine into the JPF-VM without any dynamic symbolic execution capabilities. Moreover, propagation of the taint container is only implemented once for all taint colors. Splitting the taint colors from the symbolic annotations is a clear benefit in direct comparison to the ConsiDroid [61] approach. This is the only other approach I am aware of for combining symbolic execution and dynamic tainting for JVM. As ConsiDroid represents the taint mark and the symbolic annotation with one mark, sanitization is impossible.

Taint Containers. Jaint tracks taint inside so-called taint containers. Listing 5.2 shows the example TaintContainer used in the evaluation of Jaint on the OWASP Benchmark in Paper I [107]. The different taint colors are added to the taint container as Boolean fields. Jaint generates these fields from the taint configurations. During the injection phase, if no taint container is yet attached, an instance of this taint container class is attached to a concolic value as an annotation. Then the corresponding flag is set to true, adding the taint mark on the value.

Taint Injection. Jaint injects taint in the execution on return of a source method. Thus in Listing 5.1 the getHeader method is invoked on an instance of type *HttpServletRequest.* In JPF-VM, this creates an interceptable message exit event. Jaint uses a listener for this event to access the taint container and set the *isSQLInjectionTainted* flag to true. For the presented example, the *isCrossSiteScriptingTainted* flag is set to true on the same event.

Explicit Taint Sanitization. Like taint injection, explicit taint sanitization uses the method return events. The only technical step for reflecting the sanitization effect on current execution is removing the taint mark for the color sanitized in the taint container. Technically, explicit sanitization is the inverse of taint injection, so they are quite similar in technical implementation.

Propagating Taint Containers. The taint container is associated with the values on the stack or the heap in the same way as symbolic annotations. Therefore, they travel across the JVM in the same way as symbolic annotations. Only those bytecodes that transform two values into a single new one need propagation implementation, e.g., *iadd*. For tainting in the OWASP benchmark, propagating taint along the string operations is more important. Thus Line 10 in Listing 5.1 concatenates different string parts. The Java runtime maps the concatenation to the *append* method defined in the StringBuilder class in JVM[5]. If only the string builder or the parameter has a single taint container, the result of the *append* method gets the taint container assigned. If both values have a taint container, the *combine* method on the TaintContainer class shown in Listing 5.2 merges the two containers in the resulting container; as propagation is implemented on the taint container level, they are independent of the concretely used taint colors.

Taint Checking. Jaint checks the taint marks on the parameters of a method call before invoking the method. JPF-VM creates an event for every method call. Jaint intercepts these events and checks the method name for any matching protected sink rule. If this is the case, the parameters passed to the method call are checked for set taint colors. This is done by first checking the parameter for the presence of a TaintContainer instance. The value representing the taint color of interest is checked to determine whether the container is present. If a violation is detected, the analysis records a property violation. Jaint currently stops the analysis execution if a property violation is detected. In Listing 5.1, the method call event for the prepareCall method matches the sink list for the SQL injection specification, and the parameter check is performed.

Concrete Configuration Checks. Other kinds of analysis aiming at checking concrete configuration values are typically implemented as static analyses. Adding these concrete

[5] *Remark:* At least in our implementation it does, and if the JVM is configured this way. For simplicity, I will not discuss this in detail here. The modern JVM has alternatives to interpret the string concatenation in other ways, but until recently it has always been possible to configure the JVM to map it to the *append* method of a StringBuilder.

configuration checks as an additional monitor to the dynamic taint implementation is possible using the taint check architecture. At the same places where the taint analysis checks the taint state, it is also possible to check the concrete value of an argument. For example, some security weaknesses like the use of a weak hashing function (cf. CWE-328[6]), are prevented by checking whether the constants used for configuring the function are chosen within certain allowed limits. This fundamental feature allows Jaint to solve the OWASP-Benchmark even though not all CWEs used in the OWASP-Benchmark are within the specific target domain of explicit dynamic taint analysis; hence it answers DQ1.

Discussion. Standardizing the interface for taint injection, taint removing, taint propagation, and taint checking allows the effects of a taint analysis to be generated into the concolic executor. This technical architecture enables Jaint's configuration language and answers DQ2, as it explains how Jaint allows flexible taint tracking.

We built Jaint earlier than GDart during my thesis work: it is built on top of JDart and the JPF-VM. But as demonstrated in section 3.2, most of the concepts originally designed for the concolic execution of JDart work identically in the SPouT executor tool, the concolic executor of GDart. Hence, I am confident that multi-color taint analysis as described for Jaint can be integrated into SPouT. But the proof of the concept is left open as future work on the road to scaling Jaint on top of GraalVM. The concrete explanations made in this chapter are therefore only valid for implementation on top of JPF-VM.

[6] `https://cwe.mitre.org/data/definitions/328.html` {last accessed: February 2022}

6 Evaluation and Discussion

n the previous four chapters, I have presented the key ideas that contributed to the design of JAINT and its components. This chapter describes the experiments run for empirical evaluation of the tool, and—whenever data is available—compares the results achieved with the state of the art. I will discuss the results in the second half of the chapter; this includes asking how the proposed solutions contribute to answering the initial three research questions. Seven evaluation questions structure discussion of the results. I will explain their motivation before going on to discuss the experiments and their data.

In Section 2.3 and Section 2.4, I described different strategies for improving interaction between the SMT solver and dynamic symbolic execution. Especially the recall performance of JAINT (part of DQ3) is closely coupled with the performance of the SMT solving layer, as an incomplete search is more likely to miss a weakness. In order to discuss the impact of the tool in detail and how the proposed solution contributes to recall performance, the thesis will address the following two evaluation questions:

EQ1: What are good strategies for reducing the impact of the SMT solver on the performance of a dynamic symbolic execution engine?

EQ2: Do we need SMT meta-solving strategies for SMT-based tools?

EQ1 evaluates the impact of the model generated by the SMT solver for the overall performance of dynamic symbolic execution. A smaller model makes concrete execution easier. Implicitly, this also reduces the number of propagation steps in the dynamic taint analysis if the execution path gets shorter due to the smaller model. The evaluation will show that the bounding strategy allows JDART to explore more examples successfully in the SV-COMP 2022 JAVA track; and more explored paths implies more chances for discovering a security weakness. The improved chances of discovering weaknesses increases the recall of JAINT. The answer to EQ2 discusses the dependency between a single SMT solver and the performance of the dynamic symbolic execution engine. It shows that SMT meta-solving strategies allow better tool performance compared with running a single solver. These empirical evaluations complement the discussion on the precision of JAINT in Section 5.1, with empirical measurements for improvements in the dynamic symbolic execution component that enhance JAINT's recall values.

For analyzing JAVA web applications with the combination of dynamic symbolic execution and multi-color taint analysis, the support of JAVA string operations in the dynamic symbolic execution component is essential. Moreover, there is the question whether the combination of dynamic symbolic execution and multi-color taint analysis has benefits over combining random testing with multi-color taint analysis. The following two evaluation questions address these two aspects:

EQ3: What is a promising strategy for encoding string operations in SMT-Lib during dynamic symbolic execution?

EQ4: Is it worth using dynamic symbolic execution as a path enumerator in the analysis rather than random fuzzing?

EQ3 and EQ4 evaluate the requirements for the dynamic symbolic execution engine discussed as part of DQ2. The prototype of JAINT itself already demonstrated that it is possible to combine both analyses in the same runtime, the other aspect contained in DQ2. The following two research questions evaluate the configurability of JAINT in answer to DQ1, and the overall impact of JAINT:

EQ5: How good is JAINT in comparison with the current state of the art?

EQ6: Is JAINT sufficiently configurable for the OWASP Benchmark?

The answer to the previous six evaluation question allows one to answer DQ1, DQ2, and DQ3 with empirical data extending the related theoretical considerations made in Chapter 5 regarding these questions. The data presented justifies the claim that the proposed solution answers the main research question of this thesis. For completion of the discussion as to how this thesis and the design of JAINT contribute to the overall research vision, one final evaluation question has to be discussed here:

EQ7: Does JAINT scale on real world applications?

6.1 Empirical Experiments

Four new experiments provide data for discussing the evaluation questions. These experiments are not reported in any previous paper for this thesis. In addition, the evaluation cites the results for the JAINT framework on the OWASP Benchmark from Paper I [107] as a demonstration example of JAINT's performance.

The new experiments run the GDART and JDART versions included in the SV-COMP 2022 reproduction package [21] as described in the data availability statement of this thesis. Other tools are also taken from this reproduction package. All the experiments have been run in our group's verifier cloud at TU Dortmund. This has different machines: 10x AMD Ryzen 5 PRO 4650G with 16 GB RAM, 2x Intel Core i9-10850K with 128 GB RAM, and 1x Intel Core i9-7920X with 128 GB RAM. The experiments use all the machines that fit the job, resulting in a heterogeneous machine park. If not stated otherwise, the experiments apply the same resource limits as for SV-COMP: 15 Minutes CPU time, 8 cores, and 15 GiB RAM.

Subsection 6.1.1 presents two experiments related to SMT solving. The first of these measures the impact of the bounding heuristic presented in Section 2.3; the second measures the effect of portfolio solving as presented in Section 2.4. Subsection 6.1.2 presents an experiment designed for evaluating the encoding capabilities of GDart and JDart regarding symbolic string operations as discussed in Chapter 4. Subsection 6.1.3 evaluates Jaint's performance and presents an experiment for evaluating the efficiency of random testing versus dynamic symbolic execution on the SV-COMP 2022 Java track.

6.1.1 The SMT Solving Layer Performance

Chapter 2 describes two theoretical ideas for improving the collaboration between dynamic symbolic execution and SMT solvers: bounding heuristics and meta-solving strategies. This subsection evaluates both ideas empirically. In addition, it evaluates the influence of the search bound presented in Section 3.2 as a configuration parameter in the symbolic explorer.

Bounding Heuristics. Bounding heuristics have a positive influence on the model selection problem. Figure 6.1 describes the effect of using the custom bound—as used in SV-COMP 2022 in combination with CVC4 in version 1.8, and Z3 in version 4.8.12 —as the solver backend in JDart on the SV-COMP 2022 task set consisting of 586 different verification tasks. It is clear that the bounding heuristic has a positive influence on the performance of JDart on the task set.

In the case of CVC4, the bounding heuristic masks a problem in implementation, leading to false positive results. Thus the benefit seems significantly better in this figure. However, even ignoring the effect of the incorrect verdict by excluding these two tasks, CVC4 reaches 682 points on the task set, and CVC4 with bound reaches 710 points on the task set; an improvement of 4 percent.

For Z3, performance increased from 701 points to 729 points, close to the 4 percent improvement of CVC4. Another effect visible for Z3 iis that the CPU time per problem is lower on the right hand side of the figure compared to unbounded execution. This represents the desired effects of the solver selection heuristic in practice, and confirms the expected results. Using the bounding heuristic to influence the model selection problem

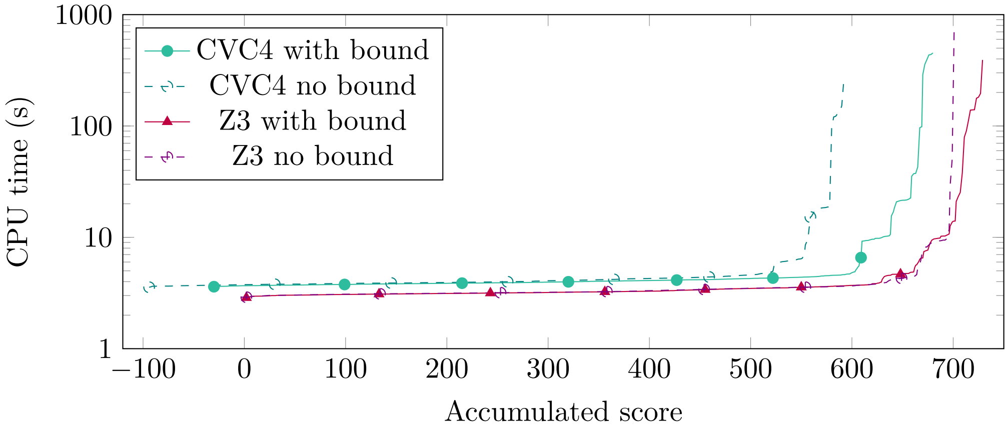


Figure 6.1: Demonstration of the impact of bounded solving heuristics on JDart's performance on the SV-COMP 2022 benchmark set.

has a positive effect on tool runtime. In the case of Z3, it not only allowed more problems to be solved. The longest-running problem reduced runtime from 788 s CPU time to 391 s CPU time.

Table 6.1 further shows the impact of different strategies for sequences of different bounds used to limit the problem. In Paper II [103], we proposed either the Fibonacci sequence or a linear bounding process. This experiment uses six values for bounding the SMT problem. The Fibonacci strategy sets out to bound the problem with the first six distinct numbers of the sequence greater than zero. The linear bound uses the values: 200, 400, 800, 1000, 1200, and 1400 as bounds. The custom bound is a sequence combining the two other strategies: 2, 8, 13, 21, 200, 600. JDart uses it for SV-COMP in 2021 and 2022. All bounds have in common that they positively impact the tool's overall performance and are recommendable in comparison with the baseline.

For CVC4, the linear and custom bound solve 520 out of 586 tasks with one incorrect answer. The linear task bound solves two more correct true cases, while the custom bound solves two more correct false cases. The Fibonacci bound solves 513 tasks with one incorrect answer. Without the bounding heuristic, CVC4 solves 498 out of 586 tasks correctly and three incorrectly. In relative terms, the linear and custom bounding heuristics together with CVC4 solve 22 tasks more than CVC4 alone, but the Fibonacci bounding heuristic solves only 15 tasks more than CVC4 alone. The margin between the different bounding heuristics is 7 tasks.

For Z3, the result is similar. The custom bounding heuristic solves the most correct false tasks with 325 to 324 found by the two other strategies. Z3 alone solves 321 correct false tasks. In total, the number of correct tasks increases from 511 for Z3 alone to 523 correct tasks for Z3 combined with Fibonacci bounding, 526 tasks for Z3 combined with

Solver	CVC4					Z3					Multi				
Bound	no	no	fib.	linear	custom	no	no	fib.	linear	custom	no	no	fib.	linear	custom
Search bounds	no	yes	yes	yes	yes	no	yes	yes	yes	yes	no	yes	yes	yes	yes
correct true	187	190	185	194	192	190	190	199	202	202	189	194	189	195	191
correct false	308	308	328	326	328	321	321	324	324	325	326	326	329	330	330
incorrect true	3	3	1	1	1	0	0	0	0	0	0	0	0	0	0
incorrect false	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
unknown	88	85	72	65	65	75	75	63	60	59	71	66	68	61	65
score	586	592	666	682	680	701	701	722	728	729	704	714	707	720	712

Table 6.1: The impact of different bounding strategies on different solving backends used in JDart. JDart's implementation is the same as the SV-COMP 2022 binary [21] using the same binary versions for the solver as contained in the official SV-COMP 2022 archive. The JConstraints layer is changed to allow additional configurations required for running the experiments. The search bounds row demonstrates the impact of bounding the symbolic search space used by JDart for SV-COMP as explained in Paper II [103]. The experiments are run with a resource budget of 15 GB RAM, 8 cores, and 15 min CPU time.

linear bounding, and 527 tasks for Z3 combined with custom bounding. The margin between the best bounded solving heuristic and plain Z3 is 16 tasks, six tasks smaller than the margin between the best bounded solving heuristic and plain CVC4.

Meta-Solving Strategies. We implemented the verdict-based second attempt strategy running CVC4 first and Z3 as the second solver as an SMT backend for JDart. The resulting strategy solver is called Multi in the JConstraints project. Ever since SV-COMP 2021, this has been the standard solver used for our dynamic symbolic execution engines. As part of the thesis, I have run the same experiments for the Multi strategy on its own and in combination with the bounding heuristic as the backend of JDart on the SV-COMP 2022 Java task set. The results are shown on the right of Table 6.1.

The plain Multi strategy solves 520 tasks correctly and therefore outperforms CVC4 (498 correct tasks) and Z3 (511 correct tasks) for correctly solved tasks. Like Z3, the Multi strategy solves none of the tasks with an incorrect verdict. The price is an increase in CPU time not reported here in total numbers. The Multi strategy works best with the linear bounded solving heuristic. It solves 195 correct true tasks and 330 correct false tasks. These make 525 tasks in total. Overall, the Multi strategy outperforms the bounded versions of CVC4 and Z3 in the number of correct false solved tasks, but Z3 solves seven more correct true tasks than the Multi strategy.

Bounded Search Space. As explained in Section 3.2, we introduced a search bound for maximum depth of the symbolic search tree. Table 6.1 shows the search bound's impact compared with plain performance. The bound influences only the correct true cases but has a negligible impact on the overall result. The difference is 3 tasks for plain CVC4 and 5 tasks for the Multi strategy. For Z3, we cannot observe any impact at

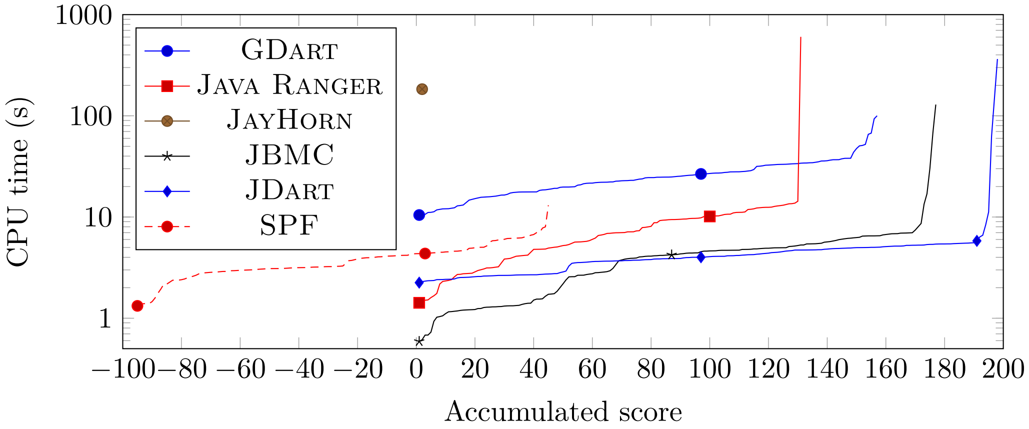


Figure 6.2: The performance of the different Java tools participating at SV-COMP 2022 on the task set containing nondeterministic string values. We used the competition configurations for the tools.

all. The depth bounds are the same as in SV-COMP 2022.

6.1.2 Empirical Evaluation of String Encodings

As adding support for string operations to JDart and GDart is one of the main contributions of this thesis, I will describe next how the new string theory implementation in JDart and GDart performs against other verifiers participating in SV-COMP 2022.

Figure 6.2 shows the result of running different Java verifiers with the artifacts that participated in SV-COMP 2022 on the subset of tasks that involved a nondeterministic string. Tasks that only involve deterministic strings or no strings are not interesting for analyzing the performance of symbolic encoding of string operations. Applying this filter, the 586 tasks in the SV-COMP 2022 Java track reduce to a subset of 184 tasks used in the presented experiment. COASTAL has been excluded from the Figure as it reported 92 incorrect true answers, leading to a score of less than -2000. Moreover, the tool runs as part of the *hors concours* without active maintenance (cf. the official SV-COMP 2022 report [18]). Hence I exclude it from further comparison among tools, although it is part of SV-COMP 2022.

Of the remaining tools, it is evident that JDart is the leading tool on this subset, and GDart is the third-best tool. JBMC lies somewhere in-between. Java Ranger is the fourth-best tool. SPF is the only tool that reported false positives on this subset of tasks. An outlier in the data set is the performance of JayHorn. While Shamakhi et al. [131], in their competition contribution on JayHorn for SV-COMP 2021, explicitly described the support for encoding string constraints in horn clauses, the tool solves only one instance of the task set that contains random (or nondeterministic) string values as input. As JayHorn encodes every Java program to a horn clause encoding, the tool may also be able to solve some of the Java programs without random inputs using strings. However, reasoning on programs of this kind has questionable value, as a program without random inputs has only a single execution path. For evaluating whether all assertions in the program hold on this single path, running the program with the JVM assertion check is sufficient. The check executes error reachability analysis for deterministic programs with the same explanatory power as any ideal verifier.

	GDart	JDart	JayHorn	SPF	Java Ranger	JBMC		
						SV-COMP	Java Test	Plain
correct true	22	31	1	20	21	25	0	25
correct false	113	136	0	101	89	127	45	115
incorrect true	0	0	0	2	0	0	0	2
incorrect false	0	0	0	2	0	0	0	0
unknown	49	17	183	59	74	32	139	42
score	157	198	2	45	131	177	45	101

Table 6.2: These are the detailed results of the experiments shown in Figure 6.2 regarding the support of symbolic string operations. The tools are run on 184 / 586 tasks of the SV-COMP 2022 Java track that involve nondeterministic strings.

Table 6.2 presents additional details on how the different tools perform on the task subset. Java Ranger is built on top of the SPF infrastructure but modifies string support in the tool. To the best of my knowledge, these modifications have not yet been described in any publication. We see that, in direct comparison, this tool solves fewer tasks than SPF but does not reach any incorrect verdict. SPF solves 121 tasks correctly but gives four incorrect answers, whereas Java Ranger solves 110 tasks correctly and gives zero incorrect answers. Given the SV-COMP points scheme, this leads to a higher score. In direct comparison with the implementation of the string theory encoding powering GDart and JDart in this thesis, Java Ranger (21) solves as many correct true cases as GDart (22), but only two-thirds as many as JDart (31). In contrast, compared with Java Ranger (89), GDart (113) solves roughly 25% more correct false tasks, and JDart (136) solves more than 50% more correct false tasks. In the category of tasks with symbolic strings, GDart and JDart outperform Java Ranger in the overall result with 135 correct tasks for GDart, 167 correct tasks for JDart, and 110 correct tasks for Java Ranger.

Comparing the string theory encoding presented in this thesis with JBMC is more complicated, as the JBMC SV-COMP competition artifact wraps the actual tool in a wrapper script executing a deterministic Java test first, with the result that the JBMC result reported by the SV-COMP artifact is a mixture between simple Java assertion tests and the verification results of JBMC itself. If the Java assertion test fails already using a deterministic test string, JBMC is not invoked at all. While this is within the rules of SV-COMP, it does not allow a fair comparison of the various encodings' capability. To clarify the influence of these different parts, Table 6.2 shows three columns for JBMC: the result of the SV-COMP artifact (also shown in Figure 6.2), the JAVA test used as the first step, and the pure JBMC tool. Compared to the implementation used in GDART, the SV-COMP artifact is better, but the pure JBMC tool performs comparably. GDART solves five tasks less than pure JBMC, but JBMC gives two incorrect true answers. In the SV-COMP artifact, the JAVA test step masks these incorrect answers. The incorrect answers require support for symbolic deletion operation on the *StringBuilder* class in one case and pattern matching with groups in the other (see Listing 4.6 for the concrete example). JBMC does not support these operations symbolically. The *StringBuilder* operations are covered in the string theory encoding presented in this thesis, and GDART and JDART solve this example flawlessly. As described above in Section 4.4, pattern matching with groups is more complex, and is listed as an open challenge for string theory encoding; GDART does not support it. For JDART, we have started to work on a search method following the considerations presented in Section 4.4, which allow exploration of such examples. But this is not suitable for the general case yet, as it does not terminate in itself and will enumerate a potentially infinite search space. However, for this concrete example, the heuristic finds the error despite the unbounded search space.

In total, GDART solves 135 correct tasks, pure JBMC solves 140 correct tasks, JBMC in the SV-COMP version solves 152 correct tasks, and JDART solves 167 correct tasks. JDART's lead over JBMC in the SV-COMP version is strongly influenced by the preliminary approach for linking numbers and strings explained at the end of Section 4.4. The detailed experiment data shows that at least in 9 of the 15 cases this preliminary approach for linking numbers and strings is responsible for the difference.

6.1.3 Evaluation of Jaint's Performance

To evaluate the performance of JAINT, we ran it on the OWASP benchmark as part of Paper I [107]. The thesis will represent this experiment. The thesis presents this experiment. In addition, I ran a second experiment that measures the performance of a pure dynamic symbolic executor vs. a random tester, in order to find assertion violations in the SV-COMP 2022 JAVA task set. Both experiments are described in this subsection.

Jaint's performance on the OWASP Benchmark. The OWASP Benchmark[1] is a collection of 2740 JAVA servlets modeling the backend of a JAVA web application. It is a project of the OWASP Foundation led by David Wichers, who created it as a

[1] `https://owasp.org/www-project-benchmark/` {last accessed: March 2022}

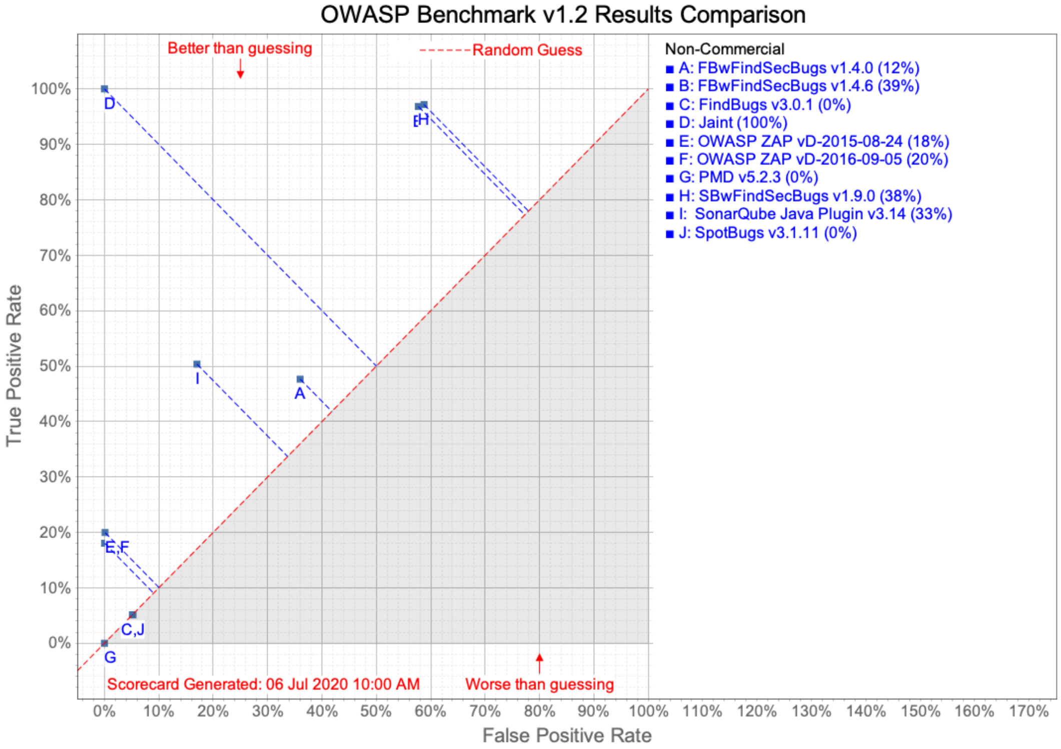


Figure 6.3: Performance of different open source SAST (A, B, C, H, I, J, G) and DAST (E, F) tools on the OWASP Benchmark. Jaint (D) is newly proposed as part of this thesis.

comparison benchmark for different security analysis tools used in the industry. Each of these servlets is classified as a representative example of one of eleven CWE classes[2]. It either contains the weakness associated with the CWE class or is secure.

Figure 6.3 shows the performance of different open source tools on the OWASP Benchmark. The y-axis lists the true positive rate. This measures how many of the contained security weaknesses a tool found. The x-axis track the false positive rate. These are tasks that the test has incorrectly qualified as a security weakness. A perfect result is the top left corner, where Jaint found all results without raising a single false positive alarm. In contrast, the bottom right corner is the worst tool, raising only false positive alarms.

Static analysis tools (SAST) are fast, as they only analyze source code without exe-

[2] Remark: CWE classes describe and index different security weaknesses and seek to classify them in a uniform database: `https://cwe.mitre.org`. Mitre runs this, and it contains 922 different weakness descriptions. SQL injection, e.g., is the entry CWE-89, a specialization of data query logic injection CWE-943.

cuting the servlets, but they report false positives. The examples included in Figure 6.3 for static analysis tools specialized in security testing are different FindSecBugs versions (A, B, H) and the SonarQube Java plugin (I). In general, we can observe a better true positive rate than for static analysis tools without security checks (e.g., SpotBugs (J) and FindBugs (C)), but they also have significant false positive rates. For the commercial static analyzer Julia [136], the authors report reaching a 90% score, including 116 false positives on this benchmark set. As it is a commercial tool, it cannot be included in the scorecard, but it would move close to the ideal top left spot. To the best of my knowledge, no static analyzer is available that solves this benchmark set without false positives.

Dynamic analysis tools (DAST), on the other hand, are slower, as they have to execute the tool under analysis. Without any knowledge of the source code internals, they will not find all security weaknesses. Nevertheless, every found weakness is a true positive case, and the design of these tools precludes false positive reports. We see this in the data reported on the OWASP benchmark for two versions of the OWASP ZAP tool (E, F).

The interactive analysis tools (IAST) by Hdiv and Contrast security bridge this gap reporting all security weaknesses in the code and no false positives leading to a perfect score of 100%. As these are commercial tools, the performance data for the scorecard is not available for inclusion in Figure 6.3. It would be the same as tool D. Both IAST tools work as monitors reporting observed security weaknesses detected during the execution of the program. As a second component they require a crawler that invokes all test cases and executes the vulnerable paths. The OWASP Benchmark provides such a crawler for analysis, but this is a downside in real-world applicability. Moreover, these monitors work by instrumenting the application and hook into the runtime. The vendors do not disclose any information regarding performance requirements or CPU time.

Dynamic Symbolic Execution vs. Random Driver. The best experiment for evaluating the design choice discussed in Section 5.2—i.e. using either dynamic symbolic execution as a path enumerator or a random path enumerator—is comparing the random path enumeration in combination with dynamic multi-color taint analysis on the OWASP Benchmark with JAINT. However, given the current implementation design of JAINT, it is impossible to run the same multi-color taint monitors with a random path driver. Instead, a proxy experiment evaluated the capabilities for finding assertion violation of JDART and GDART against random testing on the SV-COMP 2022 JAVA task set.

Figure 6.4 presents the results of the comparison between random execution of the tasks and running dynamic symbolic execution. We made three random runs with ten different random inputs (blue runs) and three different runs with 100 different random inputs (red runs). The performance reported for GDART and JDART includes reachable

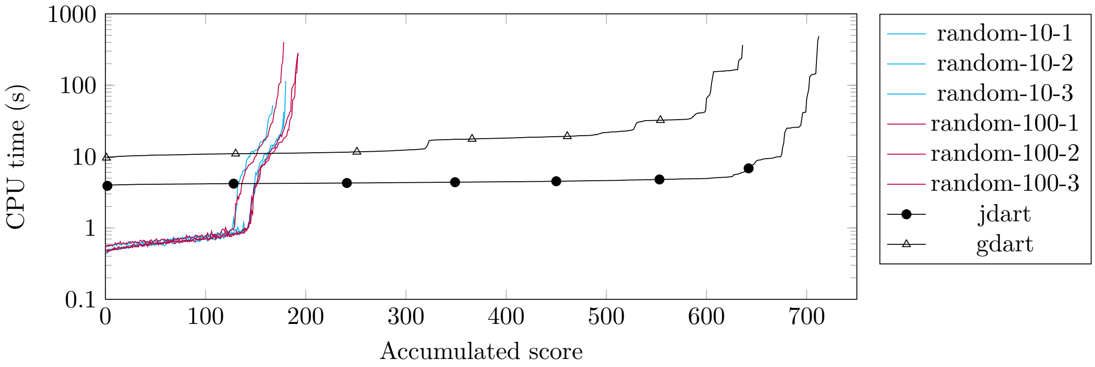


Figure 6.4: Errors found by random execution of the tool vs. guarantees given by dynamic symbolic execution.

assertions and verified absence of assertion error; thus these tools can reach more points than random testing does. I will explain these visible differences and extend them with details from the experiment: In general, executing the program with random values and checking for assertion violations in this way is significantly faster for easy cases than spinning up a new JVM for every run. Random testing spins up a single JVM and runs the test in a loop. Next, running the binary 100 times (finding 193, 193, and 194 assertion violations) evidently does not increase the number of findings significantly compared to running it ten times (finding 178, 180, and 183 assertion violations). The increase is roughly 193 to 180 tasks leading to a seven percent increase while decoupling the chance of finding an error. On the other hand, JDart found 330 (roughly + 70% compared with best random run) and GDart found 300 (roughly + 55% compared with best random run) assertion violations. Dynamic symbolic execution outperforms random testing on this task set in the number of solved tasks. In contrast, the random test found a reachable assertion violation in the *MergeSortIterative-MemSat01* example not found by any verifier in SV-COMP 2022. Java Ranger is the only verifier solving this task, claiming it to be true, which is not the case in the counterexample found by random testing.

6.2 Discussion

The previous section presented the experiments and their empirical evaluation. I will now go on to discuss the evaluation results, answer EQ7, and connect the contributions made in the thesis toward attainment of the overall research vision. Following the order in which the experiments were presented, I will first discuss the increase in SMT solving performance, before turning to the symbolic encoding of string operations, and finally the performance of JAINT. A fourth subsection will relate the discussion to the research vision of this thesis.

6.2.1 The SMT Solving Layer Performance

The implementation strategies proposed in this thesis show that SMT based analysis tools are closely linked with the performance of the SMT solver. Table 6.1 shows that exchanging the solver is sufficient to alter the performance of JDART by up to 4%. Transforming a verification problem into an SMT problem is not enough to evaluate the performance of the verification approach; investing in the interaction with the SMT solver is also required. Showing the performance of such tools on top of different SMT solvers and meta-solving strategies demonstrates the influence of the specific SMT solver.

The selection and interaction of SMT solvers in the tool's backend has a profound impact on its performance. In answer to EQ1, the implementation strategy should not simply abstract concrete implementation from a single solver. Rather the implementation strategy must optimize the resulting model for the use case, especially for tools using the resulting SMT models in concretes runs, as in dynamic symbolic execution. Using a bounded heuristic instead of a single solver for optimizing the interaction with the constraint solver can also achieve a 4% better performance. Bounding the search space has a negligible effect in the experiments compared with the heuristics and the MULTI strategy. So let's discuss the lessons learned from using heuristics and the MULTI strategy in more detail next.

Meta-Solving Strategies. Given that finding violations is a strength of dynamic symbolic execution, the experiments recommend using the MULTI strategy. JDART backed by the MULTI strategy solves more correct false tasks than any other plain solver. Exploring the complete search space is the weakness of dynamic symbolic execution, and the MULTI strategy does not significantly explore fewer problems on the SV-COMP examples than CVC4 alone. I expect that with more CPU time, the gap to Z3 will get smaller, but I have no data to prove this. By design, the MULTI strategy is more expensive than running Z3 alone. Therefore, the MULTI strategy is expected to explore less of the state space than an optimal Z3 run. The MULTI strategy only delivers value over a single solver if the solver at some point reports unknown for an SMT problem and another solver performs the reasoning on the problem. Otherwise, it is hard for the MULTI strategy to become better than a single involved solver. Overall, the evaluation shows the strengths of the meta-solving strategy, as performance increases and decreases happens at the expected interaction points. This demonstrates that the design works. Together with the data presented in Paper IV [100], these experiments show that meta-solving strategies lead to more reliable backends and are, in the worst case, on par with any single involved solver in the strategy. Nevertheless, the answer to EQ2 should be further differentiated: With SMT-based tools that challenge SMT solvers up to the point where they report unknown results, or timeouts occur, SMT meta-solving strategies have a positive impact. On the other hand, SMT-based tools that only make decisions for SMT problems easily solved by all available SMT solvers for building a meta-solving strategy will not benefit from the meta-solving strategy.

Using meta-solving strategies is not a new idea, e.g., Klee supports them [117]. However, to the best of my knowledge, we presented a design pattern for meta-solving strategies for the first time and have shown that using the earliest verdict pattern is not always the best strategy, depending on the context (cf. Paper IV [100]). The considerations for optimizing the resulting model for satisfiable instances are quite specific for tools mixing the dynamic and symbolic domains. This thesis has discussed the issue in greater detail for the first time in the context of dynamic symbolic execution, after introducing it briefly in Paper III [102].

Bounded Heuristics. The experiments applying the bounding heuristic presented in Table 6.1 show that bounding positively affects the performance of JDart. The bounding has more impact if CVC4 rather than Z3 is used as a solver in the backend. Bjørner et al. [26] maintain that models with small values are desirable for the application in programs like PEX. Thus Z3 seems partially optimized for generating small values in the model, but I have not found a detailed explanation of this property for other theories than the string solver of Z3. However, given occasional references in papers indicating that the design of Z3 already optimizes in the same direction as the bounded heuristic, it is not surprising that the bounded heuristic has more impact on the models generated by CVC4.

The experiments and work in this thesis show that additional constraints for shaping the model produced by an SMT solver positively influence the overall tool performance of JDart, although they extend the decision problem. To the best of my knowledge, this aspect is not widely discussed in the literature in relation to combining dynamic executions with SMT solving. The bounding heuristic is one strategy proposed in this thesis for decoupling the impact of a single solver on overall performance in answer to EQ1. As shown in Table 6.1, CVC4 with the custom bound solves 520 tasks correctly and one incorrectly, Z3 with the custom bound solves 527, and the Multi strategy solves 521 tasks correctly. These results are more homogeneous than those using the plain solvers: CVC4 solves 498, Z3 511, and the Multi strategy 520. Hence the bounding heuristic has not only improves the overall performance but also reduces the impact of the specific SMT backend on the performance of the dynamic symbolic execution engine.

6.2.2 DSE as Path Enumerator for Jaint

As described in Section 5.2, JAINT uses dynamic symbolic execution as path enumerator. The integration of encoding for string operations is crucial for the analysis of web applications. Moreover, this thesis raises the question whether dynamic symbolic execution is worth the resource investment or whether random path enumeration is better in practice. This subsection will discuss these two aspects.

String encoding. As described in the thesis, analyzing string operations in path constraints is especially important for analyzing JAVA Web applications. In Chapter 4the thesis has presented two different encoding strategies:: $\mathcal{SL}_{\mathcal{BV}}$ and $\mathcal{SL}_{\mathcal{SMT}}$. During the design phase and in the context of SV-COMP, the string theory encoding $\mathcal{SL}_{\mathcal{SMT}}$ has already proven to be more expressive and powerful than the bitvector encoding $\mathcal{SL}_{\mathcal{BV}}$. Purely from the design perspective, then, the answer to EQ3 is clearly the string theory encoding.

The experiment in Subsection 6.1.2 shows that the string theory encoding implemented in JDART is currently the most advanced existing implementation in a JAVA verifier that has participated in the SV-COMP java track. The implementation in GDART is performance-wise comparable with the implemented encoding in JBMC. The evaluation shows that the symbolic encoding of operations in the bounded model checking component of JBMC does not solve significantly more tasks correctly than GDART. However, in SV-COMP, JBMC is paired with a testing component for overall verification. The combined approach of bounded model checking and testing ranks performance-wise between GDART and JDART. This empirical result confirms the potential of string theory encoding as a promising strategy.

Random Driver vs. Dynamic Symbolic Execution. The experiments presented on the OWASP benchmark in Subsection 6.1.3 demonstrate that the taint monitors work, and multi-color taint analysis contributes to the proposed analysis by detecting security weaknesses. The dynamic symbolic execution engine also enumerates the paths in the benchmark. Nevertheless, it does not answer whether using dynamic symbolic execution as path enumeration is worth the cost of running the analysis. If the given use case does not require guarantees on the absence of security weaknesses, pairing the taint analysis with a random path driver will in theory also finds security weaknesses. The question is whether this is cheaper than running dynamic symbolic execution.

The experiment comparing random testing for assertion violation detection with dynamic symbolic execution has shown that dynamic symbolic execution is more expensive but delivers better results than random testing. Summing this up as an answer to EQ4: Given the experiments, dynamic symbolic execution is more powerful than random testing for driving the analysis. Sometimes, when dynamic symbolic execution exhausts its resource limit without finding a vulnerability, giving a random driver a second chance will allow some otherwise unnoticed errors to be caught. Nevertheless, the long-term goal for JAINT is a powerful dynamic symbolic execution engine that drives path exploration, and the experiments confirm this design decision.

As JBMC has shown, combining testing with verification is sometimes more powerful than applying a single technique alone. If sufficient CPU and RAM resources are available, combining random testing and dynamic symbolic execution as a driver for the multi-color taint analysis is also promising. The modular design supports this with a few changes to the driver that replace the *Verifer* class with a random input generator. The taint analysis and configuration stay untouched.

6.2.3 Jaint's Performance and Scalability

To the best of my knowledge and as shown in the scorecard in Figure 6.3, JAINT is the first research tool that reports a perfect result on the OWASP Benchmark, demonstrating the potential of the approach. The fact that JAINT has analyzed the complete benchmark successfully demonstrates that it has been possible to encode all eleven CWEs included in the OWASP Benchmark with JAINT's configuration language. The proposed taint tracking architecture worked for these CWEs. As described in Section 5.4, some of the involved CWEs in the OWASP Benchmark work in the form of static value checks rather than taint flow properties. The JAINT taint tracking architecture is also flexible enough to describe these CWEs. This answers EQ6 affirmatively.

I will now briefly discuss JAINT's performance and design, compared with other industrial tools reporting results on the OWASP Benchmark, and address the scalability of JAINT.

Performance. The JAINT framework presented here also falls into the category of an IAST tool. Compared with the solutions by Hdiv and Contrast security, integrated dynamic symbolic execution allows all paths to be explored without requiring the external crawler component. Hence the reported total score for all three tools is the same, but JAINT is the first tool that includes path enumeration as well as monitoring in the analysis. All IAST tools outperform DAST and SAST open source solutions as well as the state-of-the-art commercial analyzer Julia for this benchmark. In sum, JAINT not only demonstrates a promising taint engine that allows state-of-the-art security monitoring but allows completely automated analysis not supported by the current state of the art. This automated analysis is an explicit improvement compared to existing tools, and represents my answer to EQ5.

Scalability. The previous experiments have shown that JAINT and its subcomponents work well on the existing benchmarks, but what are the expectations for using JAINT on real-world software? As JAINT in its current implementation uses JDART and the JPF-VM, it is impossible to run it on arbitrary software as the JPF-VM does not support the complete JAVA standard library: for example, the JPF-VM does not support central parts of the JAVA io and nio library, making it impossible to analyze problems involving file operations. With GDART, we started to replace JPF-VM with GRAALVM in the JAINT framework. In this thesis, the step of migrating the multi-color taint monitors is left for future work. With the experiences gathered during the implementation of GDART, I expect the whole design of the JAINT framework to transfer to GRAALVM. So far, all concepts relevant for implementing dynamic symbolic execution in JPF-VM have direct conceptual counterparts in the GRAALVM. As the multi-color taint analysis uses the same implementation strategies, there is no reason why the implementation of multi-color taint analyses should not transfer in the same way. By encoding string operations symbolically, improving the solving layer, and discussing the capabilities of mocking and modeling the execution environment for the analysis, this thesis has addressed the conceptual challenges identified for scaling JAINT and demonstrated them in a prototype. Migrating the prototype implementation to GRAALVM is currently a work in progress. Answering EQ7 in the long-term requires more real-world experience with the tool, resulting in completing migration from the JPF-VM to the GRAALVM.

6.2.4 Contribution to the Research Vision

Especially the answer to EQ5 has shown that combining dynamic multi-color taint analysis with dynamic symbolic execution allows analysis of JAVA web applications security. The promised goal for this thesis has been reached by the different contributions mentioned in the thesis. The remaining section will discuss how this contributes to the research vision.

The answers given to DQ1 have demonstrated that the general taint tracking framework is tailorable for different analyses that run in parallel but always analyze exactly one execution path. The limitation to a single path contributes significantly to the precision of the analysis, as a violation of a taint property is always detected on precisely one path. The concrete driver generates the concrete values that allow reproduction of the path that has violated the property. Concrete values are a good starting point for working in future toward explanations for security weaknesses and tool integrations that guide human attention to eliminating these from the source code. The approach in itself already delivers information in this direction.

The existence of the JAINT prototype itself has shown that the proposed answer to DQ2 is feasible. Implementing dynamic symbolic execution and multi-color taint analysis within the same runtime is possible. The empirical data presented in answer to EQ4 shows the value of using dynamic symbolic execution instead of random testing as a driver for taint analysis. Dynamic symbolic execution is often seen as inefficient, as it suffers from the state space explosion problem. Section 3.3 discusses how different strategies for choosing the driver method for the system under analysis help diminish the influence of the state space explosion problem allowing partial analysis of a program. Of course, this does not solve the state space explosion problem but does allow practical applications of the JAINT framework.

For the precision and the recall of JAINT, the main objective of DQ3—discussion of the right concrete driver—impacts recall, as the symbolic scope is defined here. JAINT's recall values are closely coupled with the dynamic symbolic execution performance and, consequently, with the performance of the SMT solving layer used. The thesis has proposed solutions for improving the performance of dynamic symbolic execution, especially while analyzing programs with string operations (see the experiments run in answer to EQ3). Moreover, the proposed SMT meta-solving strategies and bounding heuristic reduce the dependency between dynamic symbolic execution and a single SMT solver. JDART's gold medal in the JAVA track of SV-COMP 2022 demonstrates the recall potential of dynamic symbolic execution tools on JAVA programs. Furthermore, JDART found the most assertion violations (cf. the discussion on GDART's performance compared with other tools in Paper V [101]).

JAINT is very precise. However, as discussed in Section 5.1, in its current implementation, JAINT might give imprecise answers because of implicit sanitization introduced for constant functions. This type of sanitization is currently not implemented. But as argued before, implicit sanitization of constant functions is not expected to be a common case for designed sanitization in JAVA web applications. Nevertheless, it is possible to improve implementation in this area at the cost of slower concrete execution.

Summing up the evaluation, JAINT has not proved itself yet on real-world software, for the reasons discussed at the end of Section 6.1.3. Nevertheless, the existing prototype of JAINT demonstrates its general feasibility and delivers some promising preliminary results for its potential as a candidate tool fitting the research vision. Furthermore, it provides precise counter-examples if a security weakness is found and certifies the absence of security weaknesses if the search terminates. Given the data and considerations presented, I am confident that JAINT will develop further toward fullfillment of the main research vision. In its current state it in any case already answers the main research question.

6.2.5 Threats to Validity

As in any research project, there are many threats to the validity of the results obtained in this thesis. I will explicitly discuss two threats to the internal validity of the thesis and two to its external validity resulting from the benchmark sets used there.

Internal Validity. There are two main threats to internal validity: heterogeneous machines and seedable heuristics.

The experiments use a heterogeneous park of different machines. The measurements are therefore affected by different CPU speeds in the consumed CPU time. As the argumentation mainly uses the total number of solved tasks, and most tasks contributing to the overall score are not close to the time limit as shown in Figure 6.2, the variance of CPU speed in the specific experiments presented in this thesis is negligible.

SMT solvers use internal heuristics with random choices that might influence the experiments. These random choices are especially a threat to the validity of the experiments presented in Figure 6.1. Therefore, in order to diminish the effects of different start seeds for the heuristics in multiple runs, the SMT solvers are always seeded with the same seeds.

External Validity. The experiments use two benchmark sets established in the literature. There is no literature available that shows that these benchmark sets are representative for real-world software or how the results are transferable. I will discuss the two benchmarks in more detail in the following two paragraphs.

The experiments presented in Subsection 6.1.1 and Subsection 6.1.2 use the SV-COMP benchmark for the JAVA track. While it is a best-effort community benchmark set, it has not been demonstrated that the performance a tool achieves on this benchmark transfers to real-world software. However, the SV-COMP benchmark is currently used for comparing JAVA tools, e.g., in evaluating the SYMJEX tool [83], and it is the agreed standard within the community. A performance increase demonstrated on this benchmark is expected to be also observable on real-world software with the same tendency. The same tendency means that if a solution saves time on the benchmark, the solution will also save time on real-world software.

The OWASP Benchmark used for the experiments in Subsection 6.1.3 is artificially generated and only inspired by real-world software. As the industry established this benchmark as an indicator of tool quality, solving this benchmark shows that the concept works. Transferring the results from the benchmark onto real-world software requires further proof of the concept for generalizing the results. As other tools show shortcomings and JAINT does solve this benchmark without them, it is a valid first step for demonstrating the framework's potential. But this thesis does not investigate whether these results are transferable to other JAVA programs.

7 Conclusion and Future Work

This thesis demonstrates how multi-color taint analysis interacts with dynamic symbolic execution, allowing IT security analysis of JAVA web applications. It presents the research challenge and the need to prove the absence of security weaknesses in programs and otherwise detect security weaknesses precisely. The proposed solution for JAVA web applications demonstrates an advance in this direction. In the following two sections, I will conclude the thesis with a summary of the key results and discuss open questions and ideas for future work.

7.1 Conclusion

In this thesis I have presented the JAINT framework allowing the automated analysis of JAVA web applications for security weaknesses. The JAINT framework combines dynamic symbolic execution as a path enumerator with a dynamic multi-color taint analysis for monitoring security weaknesses during execution. If a security weakness is detected, the taint analysis monitors will generate precise counter-examples that drive the execution along the path with the observed weakness. While this architecture borrows aspects of security fuzzer design, dynamic symbolic execution will guarantee that the taint analysis checks all reachable paths in the program. This guarantee is limited to cases where the dynamic symbolic exploration terminates. Therefore, if the analysis terminates, the resulting framework will verify the absence of security weaknesses in JAVA web applications. Security fuzzers do not usually give these guarantees, as they use random or semi-random driver methods. In direct comparison with DAST tools, the JAINT framework achieves better recall values, and in contrast with SAST tools, its precision is better.

Apart from the framework, the thesis has presented how to build efficient meta-solving strategies, integrate string operations in the dynamic symbolic execution of JAVA programs, and configure taint analysis using the JAINT configuration language.

In the area of SMT solving, the thesis has presented different strategies for influencing the model generated for satisfiable SMT problems, and has proposed patterns for constructing SMT meta-solvers. Dynamic symbolic execution turns the model into concrete values that impact the program's runtime during path recording. Hence the thesis has presented the model selection problem and the bounding heuristic for getting smaller models. The experiments show that this has a beneficial impact on tool performance. Further, the thesis describes a design paradigm for simulating meta-solving strategies before implementing them. The evaluation shows that the meta-solving strategy detects roughly 1.5% more correct false results than a single solver on its own in the context of dynamic symbolic execution on the SV-COMP 2022 Java task set. Furthermore, the described design patterns for combining decision engines are general enough to be transferred to other domains requiring a decision outside of SMT solving.

For dynamic symbolic execution of Java programs, the thesis contributes the capability to encode string constraints on the Java string library level. This encoding allows the symbolic exploration of string operations often used in web applications. These changes are a core requirement for using dynamic symbolic execution in the analysis of web applications. The experiments have shown that the resulting implementation for dynamic symbolic execution of Java programs leads its field.

As the third significant contribution, the thesis has implemented a multi-color taint analysis, and in the resulting paper, we have described how this analysis is configured.

These contributions together allowed implementation of the Jaint framework as a proof of concept. The resulting implementation outperforms any other research tool in precision and recall on the OWASP-Benchmark, a suite of Java servlets for comparing security analyses for Java. Jaint provides a precise witness that violates the taint policy, leading to an alarm. The high precision makes manual post-processing of the analysis result unnecessary, a step commonly required for static analyses.

Last but not least, the thesis has taken the first step toward scaling the Jaint framework from research examples to real-world applications by lifting dynamic symbolic execution implementation from JPF-VM to the industry-grade GraalVM. These efforts resulted in the introduction of GDart and SPouT. Lifting the implementation from JPF-VM to GraalVM demonstrates that the concept fits well in the JVM architecture independently of the concrete JVM implementation. Furthermore, it strengthens the argument that the problems occurring with the integration of string operations into dynamic symbolic execution are conceptual and not implementation-dependent.

The previously described contributions are the design draft for a tool that proves the absence of IT security weaknesses in Java programs. The prototype implementation provided here demonstrates the practical feasibility of the designed approach. Moreover, it represents an advance toward a development process that promises to prevent future IT security breaches.

7.2 Future Work

However, the development of the design draft and its prototype implementation have raised additional questions and the prospect of future work. Completing these additional tasks will advance further toward the research vision of a secure development process.

Scaling Jaint on Real-World Software. The evaluation in Chapter 6 shows that Jaint and its subcomponents work well on the benchmarks presented. To stimulate further research in this area and identify open challenges for scaling Jaint, analysis of real-world applications is the next step. Running these will mean lifting the complete Jaint framework from JPF-VM to GraalVM. So far, introducing GDart completes half of this task. Implementing dynamic multi-color taint analysis in SPouT is still an open task. However, completing it will allow execution of Jaint on top of GraalVM, paving the way for an extensive case study on real-world software. The case study is expected to generate additional insights into the remaining challenges for scaling Jaint.

Automated Generation of Analysis Drivers. The thesis has described the impact of symbolic and concrete instantiation of certain data types in the driver for dynamic analysis. This driver is often designed as a wrapper around the method under analysis. In future, a method for automated generation of driver methods that might also work in continuous integration setups offers a worthy area of research.

Using Static Analysis as Path Generator. The current design uses dynamic symbolic execution as the path generator in Jaint. In future, researching the interplay of dynamic symbolic execution and other static analysis techniques is a promising area. I expect a robust static analysis to limit the state space that dynamic symbolic execution must explore for a verification verdict. The goal is to reduce the false verdicts common in static analysis by applying dynamic analysis. At the same time, static analysis might be faster in ruling out uninteresting program areas and, therefore, may save significant resources.

Parallel Exploration. The loosely coupled architecture of GDart allows multiple SPouT instances to be starte, each exploring a different path. These parallel runs allow the execution tree to be examined using either multiple machines or serverless architectures in the cloud. Typically a single executor run requires only a few resources and finishes within minutes, while the symbolic exploration component requires more resources and has a longer overall runtime. Exploiting this potential for parallelism and adjusting it to the usecase is an open question.

Bibliography

[1] P. A. Abdulla et al. "Trau: SMT solver for string constraints." In: *2018 Formal Methods in Computer Aided Design (FMCAD).* Oct. 2018, pp. 1–5. DOI: 10.23919/FMCAD.2018.8602997.

[2] Parosh Aziz Abdulla et al. "Norn: An SMT solver for string constraints." In: *International Conference on Computer Aided Verification.* Springer. 2015, pp. 462–469.

[3] Alfred V. Aho. "Algorithms for Finding Patterns in Strings." In: *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity.* Cambridge, MA, USA: MIT Press, 1991, pp. 255–300. ISBN: 0444880712.

[4] Jon Allen. *Perl Version 5.8.8 Documentation - Perlsec.* http://perldoc.perl.org/5.8.8/perlsec.pdf. May 2016.

[5] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269. DOI: 10.1145/2594291.2594299.

[6] Thanassis Avgerinos et al. "Enhancing symbolic execution with veritesting." In: *Proceedings of the 36th International Conference on Software Engineering.* 2014, pp. 1083–1094. DOI: 10.1145/2568225.2568293.

[7] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[8] Algirdas Avizienis and Liming Chen. "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution." In: *Proceedings of COMPSAC 77.* IEEE, 1977, pp. 149–155.

[9] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. "Automata-based model counting for string constraints." In: *International Conference on Computer Aided Verification.* Springer. 2015, pp. 255–272. DOI: 10.1007/978-3-319-21690-4_15.

[10] Roberto Baldoni et al. "A survey of symbolic execution techniques." In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 50. DOI: 10.1145/3182657.

[11] Davide Balzarotti et al. "Saner: Composing static and dynamic analysis to validate sanitization in web applications." In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on.* IEEE. 2008, pp. 387–401. DOI: 10.1109/SP.2008.22.

[12] Tao Bao et al. "Strict control dependence and its effect on dynamic information flow analyses." In: *Proceedings of the 19th international symposium on Software testing and analysis.* 2010, pp. 13–24. DOI: 10.1145/1831708.1831711.

[13] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver." In: *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6.* Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[15] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-Lib standard: Version 2.0." In: *Proc. of the 8th International Workshop on Satisfiability Modulo Theories.* Vol. 13. 2010, p. 14.

[16] Clark Barrett et al. "CVC4." In: *Computer Aided Verification.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14.

[17] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. "Z3str3: A String Solver with Theory-aware Heuristics." In: *2017 Formal Methods in Computer Aided Design (FMCAD).* 2017, pp. 55–59. DOI: 10.23919/FMCAD.2017.8102241.

[18] Dirk Beyer. "Progress on Software Verification: SV-COMP 2022." In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20.

[19] Dirk Beyer. "Software Verification with Validation of Results." In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 331–349. DOI: 10.1007/978-3-662-54580-5_20.

[20] Dirk Beyer. "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)." In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24.

[21] Dirk Beyer. *Verifiers and Validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022).* Version svcomp22. Feb. 2022. DOI: 10.5281/zenodo.5959149.

[22] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable benchmarking: requirements and solutions." In: *International Journal on Software Tools for Technology Transfer* 21.1 (2019), pp. 1–29. DOI: `10.1007/s10009-017-0469-y`.

[23] A. Biere et al. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[24] Armin Biere et al. "Symbolic model checking without BDDs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1999, pp. 193–207. DOI: `10.1007/3-540-49059-0_14`.

[25] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–199. ISBN: 978-3-662-46681-0. DOI: `10.1007/978-3-662-46681-0_14`.

[26] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. "Path feasibility analysis for string-manipulating programs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 307–321. DOI: `10.1007/978-3-642-00768-2_27`.

[27] Nikolaj Bjørner et al. "An SMT-Lib format for sequences and regular expressions." In: *SMT* 12 (2012), pp. 76–86. DOI: `10.29007/w5m5`.

[28] Dmitry Blotsky et al. "StringFuzz: A fuzzer for string solvers." In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 45–51. DOI: `10.1007/978-3-319-96142-2_6`.

[29] Tim Boland and Paul E Black. "Juliet 1. 1 C/C++ and java test suite." In: *Computer* 45.10 (2012), pp. 88–90. DOI: `10.1109/MC.2012.345`.

[30] Pierre Bourque et al. "The guide to the software engineering body of knowledge." In: *IEEE software* 16.6 (1999), pp. 35–44. DOI: `10.1109/52.805471`.

[31] Martin Brain, Florian Schanda, and Youcheng Sun. "Building better bit-blasting for floating-point problems." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 79–98. DOI: `10.1007/978-3-030-17462-0_5`.

[32] L. Braz et al. "Why Don't Developers Detect Improper Input Validation? '; DROP TABLE Papers; –." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 499–511. DOI: `10.1109/ICSE43902.2021.00054`. URL: `https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00054`.

[33] Tegan Brennan et al. "Constraint normalization and parameterized caching for quantitative program analysis." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 2017, pp. 535–546. DOI: 10.1145/3106237.3106303.

[34] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. "ASM: a code manipulation tool to implement adaptable systems." In: *Adaptable and extensible component systems* 30.19 (2002).

[35] Tevfik Bultan et al. *String Analysis for Software Verification and Security.* Springer, 2017. DOI: 10.1007/978-3-319-68670-7.

[36] Jerry R Burch et al. "Symbolic model checking: 1020 states and beyond." In: *Information and computation* 98.2 (1992), pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A.

[37] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI.* Vol. 8. 2008, pp. 209–224.

[38] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later." In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: http://doi.acm.org/10.1145/2408776.2408795.

[39] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. "On the Limits of Information Flow Techniques for Malware Analysis and Containment." In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by Diego Zamboni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 143–163. DOI: 10.1007/978-3-540-70542-0_8.

[40] Taolue Chen et al. "Decision procedures for path feasibility of string-manipulating programs with complex operations." In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30. DOI: 10.1145/3290362.

[41] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. "Precise Analysis of String Expressions." In: *Proc. 10th International Static Analysis Symposium (SAS).* Vol. 2694. LNCS. Available from http://www.brics.dk/JSA/. Springer-Verlag, June 2003, pp. 1–18. DOI: 10.1007/3-540-44898-5_1.

[42] V. Chvátal. *Linear Programming.* Series of books in the mathematical sciences. W.H. Freeman and company, 1983. ISBN: 9780716711957.

[43] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs." In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[44] Edmund M Clarke et al. *Handbook of model checking.* Vol. 10. Springer, 2018. DOI: 10.1007/978-3-319-10575-8.

[45] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework." In: *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM. 2007, pp. 196–206. DOI: 10.1145/1273463.1273490.

[46] Juan José Conti and Alejandro Russo. "A taint mode for python via a library." In: *Nordic Conference on Secure IT Systems.* Springer. 2010, pp. 210–222. DOI: 10.1007/978-3-642-27937-9_15.

[47] Byron Cook et al. "Model checking boot code from AWS data centers." In: *Formal Methods in System Design* (2020), pp. 1–19. DOI: 10.1007/978-3-319-96142-2_28.

[48] Ricardo Corin and Felipe Andrés Manzano. "Taint Analysis of Security Code in the KLEE Symbolic Execution Engine." In: *Information and Communications Security.* Ed. by Tat Wing Chim and Tsz Hon Yuen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 264–275. ISBN: 978-3-642-34129-8. DOI: 10.1007/978-3-642-34129-8_23.

[49] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: Past, Present and Future." In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* CSL-LICS '14. Vienna, Austria: Association for Computing Machinery, 2014. ISBN: 9781450328869. DOI: 10.1145/2603088.2603165.

[50] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: a flexible information flow architecture for software security." In: *ACM SIGARCH Computer Architecture News* 35.2 (2007), pp. 482–493. DOI: 10.1145/1250662.1250722.

[51] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Real-World Buffer Overflow Protection for Userspace and Kernelspace." In: *USENIX Security Symposium.* 2008, pp. 395–410.

[52] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Tainting is not pointless." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 88–92. DOI: 10.1145/1773912.1773933.

[53] Ali Davanian et al. "DECAF++: Elastic Whole-System Dynamic Taint Analysis." In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses.* Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 31–45.

ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/davanian.

[54] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving." In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[55] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory." In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[56] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[57] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow." In: *Commun. ACM* 20.7 (July 1977), pp. 504–513. ISSN: 0001-0782. DOI: 10.1145/359636.359712.

[58] Marko Dimjašević et al. "Study of integrating random and symbolic testing for object-oriented software." In: *International Conference on Integrated Formal Methods.* Springer. 2018, pp. 89–109. DOI: 10.1007/978-3-319-98938-9_6.

[59] Marko Dimjašević et al. "The Dart, the Psyco, and the Doop: Concolic Execution in Java PathFinder and Its Applications." In: *SIGSOFT Softw. Eng. Notes* 40.1 (Feb. 2015), pp. 1–5. DOI: 10.1145/2693208.2693248.

[60] Claudia Eckert. *IT-Sicherheit: Konzepte-Verfahren-Protokolle.* 10. De Gruyter, 2018. DOI: 10.1515/9783110563900.

[61] Ehsan Edalat, Babak Sadeghiyan, and Fatemeh Ghassemi. "ConsiDroid: A Concolic-based Tool for Detecting SQL Injection Vulnerability in Android Apps." In: *CoRR* abs/1811.10448 (2018). arXiv: 1811.10448.

[62] Marcelo M. Eler, Andre T. Endo, and Vinicius H.S. Durelli. "An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective." In: *Journal of Systems and Software* 121 (2016), pp. 281–297. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.03.020.

[63] Michael Feathers. *Working Effectively with Legacy Code.* USA: Prentice Hall PTR, 2004. ISBN: 0131177052.

[64] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. "Tailoring taint analysis to GDPR." In: *Annual Privacy Forum.* Springer. 2018, pp. 63–76. DOI: 10.1007/978-3-030-02547-2_4.

[65] Paul Fiterău-Broştean and Falk Howar. "Learning-Based Testing the Sliding Window Behavior of TCP Implementations." In: *Critical Systems: Formal Methods and Automated Verification*. Ed. by Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti. Cham: Springer International Publishing, 2017, pp. 185–200. DOI: `10.1007/978-3-319-67113-0_12`.

[66] Gordon Fraser and Andrea Arcuri. "A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (2014), p. 8. DOI: `10.1145/2685612`.

[67] John Galea and Daniel Kroening. "The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation." In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 622–636. DOI: `10.1145/3320269.3384764`.

[68] Ashish Garg, Jeffrey Curtis, and Hilary Halper. "Quantifying the financial impact of IT security breaches." In: *Information Management & Computer Security* (2003). DOI: `10.1109/PST.2014.6890934`.

[69] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6. DOI: `10.1007/978-3-642-19237-1_4`.

[70] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." In: *Communications of the ACM* 55.3 (2012), pp. 40–44. DOI: `10.1145/2090147.2094081`.

[71] George Hagen and Cesare Tinelli. "Scaling up the formal verification of Lustre programs with SMT-based techniques." In: *2008 Formal Methods in Computer-Aided Design*. IEEE. 2008, pp. 1–9. DOI: `10.1109/FMCAD.2008.ECP.19`.

[72] Vivek Haldar, Deepak Chandra, and Michael Franz. "Dynamic taint propagation for Java." In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE. 2005, 9–pp. DOI: `10.1109/CSAC.2005.21`.

[73] William Halfond, Alex Orso, and Pete Manolios. "WASP: Protecting web applications using positive tainting and syntax-aware evaluation." In: *IEEE transactions on Software Engineering* 34.1 (2008), pp. 65–81. DOI: `10.1109/TSE.2007.70748`.

[74] Ari B Hayes et al. "GPU Taint Tracking." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 209–220.

[75] Andrew Henderson et al. "Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 2014, pp. 248–258. DOI: 10.1145/2610384.2610407.

[76] Pieter Hooimeijer et al. "Fast and Precise Sanitizer Analysis with BEK." In: *USENIX Security Symposium.* Vol. 58. 2011.

[77] Falk Howar, Fadi Jabbour, and Malte Mues. "JConstraints: a library for working with logic expressions in Java." In: *Models, Mindsets, Meta: The What, the How, and the Why Not?* 2019. DOI: 10.1007/978-3-030-22348-9_19.

[78] Falk Howar and Malte Mues. "Can We Trust Theorem Provers for Industrial AI?" In: *IEEE Software* 38.6 (2021), pp. 104–108. DOI: 10.1109/MS.2021.3103448.

[79] Andrew Hunt and Hunt Thomas. *The Pragmatic Programmer.* Addison-Wesley Professional, 1999. ISBN: 9780201616224.

[80] Joxan Jaffar et al. "TracerX: Dynamic symbolic execution with interpolation (competition contribution)." In: *Fundamental Approaches to Software Engineering* 12076 (2020), p. 530. DOI: 10.1007/978-3-030-45234-6_28.

[81] Dejan Jovanović and Clark Barrett. "Polite theories revisited." In: *Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17.* Springer. 2010, pp. 402–416. DOI: 10.1007/978-3-642-16242-8_29.

[82] James C. King. "Symbolic execution and program testing." In: *Commun. ACM* 19.7 (1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[83] Sebastian Kloibhofer et al. "SymJEx: Symbolic Execution on the GraalVM." In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes.* MPLR 2020. Virtual, UK: Association for Computing Machinery, 2020, pp. 63–72. ISBN: 9781450388535. DOI: 10.1145/3426182.3426187.

[84] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* 2nd ed. Springer Publishing Company, Incorporated, 2016. DOI: 10.1007/978-3-662-50497-0.

[85] Tammo Krueger et al. "TokDoc: A Self-Healing Web Application Firewall." In: *Proceedings of the 2010 ACM Symposium on Applied Computing.* SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 1846–1853. DOI: 10.1145/1774088.1774480.

[86] Ben Livshits. "Dynamic Taint Tracking in Managed Runtimes." In: MSR-TR-2012-114 (Nov. 2012). URL: `https://www.microsoft.com/en-us/research/publication/dynamic-taint-tracking-in-managed-runtimes/`.

[87] V Benjamin Livshits and Monica S Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In: *USENIX Security Symposium*. Vol. 14. 2005, pp. 18–18.

[88] Blake Loring, Duncan Mitchell, and Johannes Kinder. "Sound regular expression semantics for dynamic symbolic execution of JavaScript." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 425–438. DOI: `10.1145/3314221.3314645`.

[89] Kasper Luckow et al. "JDart: A Dynamic Symbolic Analysis Framework." In: *TACAS*. 2016. DOI: `10.1007/978-3-662-49674-9_26`.

[90] Linghui Luo. "Improving Real-World Applicability of Static Taint Analysis." PhD thesis. Universität Paderborn, Oct. 2021. URL: `https://www.bodden.de/pubs/phdLuo.pdf`.

[91] Muhammad Numair Mansur et al. "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing." In: *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 701–712. ISBN: 9781450370431. DOI: `10.1145/3368089.3409763`.

[92] Michael Martin, Benjamin Livshits, and Monica S Lam. "Finding application errors and security flaws using PQL: a program query language." In: *Acm Sigplan Notices* 40.10 (2005), pp. 365–383. DOI: `10.1145/1103845.1094840`.

[93] Kenneth L McMillan. "Interpolation and SAT-based model checking." In: *International Conference on Computer Aided Verification*. Springer. 2003, pp. 1–13. DOI: `10.1007/978-3-540-45069-6_1`.

[94] Jiang Ming et al. "StraightTaint: Decoupled Offline Symbolic Taint Analysis." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 308–319. ISBN: 9781450338455. DOI: `10.1145/2970276.2970299`.

[95] Jiang Ming et al. "TaintPipe: Pipelined Symbolic Taint Analysis." In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 65–80. ISBN: 978-1-931971-232.

[96] Federico Mora et al. “Z3str4: A Multi-armed String Solver.” In: *Formal Methods*. Ed. by Marieke Huisman, Corina Păsăreanu, and Naijun Zhan. Cham: Springer International Publishing, 2021, pp. 389–406. DOI: 10.1007/978-3-030-90870-6_21.

[97] Malte Mues. *Addition Reproduction Package Thesis Mues*. May 2023. DOI: 10.5281/zenodo.7944937. URL: https://doi.org/10.5281/zenodo.7944937.

[98] Malte Mues, Martin Fitzke, and Falk Howar. “Thoughts about using Constraint Solvers in Action.” In: *Electronic Communications of the EASST* 78 (2020). DOI: 10.14279/tuj.eceasst.78.1100.

[99] Malte Mues, Sebastian Gerard, and Falk Howar. “Identification of Spurious Labels in Machine Learning Data Sets using N-Version Validation.” In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. 2020, pp. 1–7. DOI: 10.1109/ITSC45102.2020.9294223.

[100] Malte Mues and Falk Howar. “Data-Driven Design and Evaluation of SMT Meta-Solving Strategies: Balancing Performance, Accuracy, and Cost.” In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 179–190. DOI: 10.1109/ASE51524.2021.9678881.

[101] Malte Mues and Falk Howar. “GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022. DOI: 10.1007/978-3-030-99527-0_27.

[102] Malte Mues and Falk Howar. “JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2020, pp. 398–402. DOI: 10.1007/978-3-030-45237-7_28.

[103] Malte Mues and Falk Howar. “JDart: Portfolio Solving, Breadth-First Search and SMT-Lib Strings.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2021. DOI: 10.1007/978-3-030-72013-1_30.

[104] Malte Mues and Falk Howar. *tudo-aqua/paper-reproduction-package-ase2021: Reproduction Package for the ASE 2021 AEC Committee*. July 2021. DOI: 10.5281/zenodo.5226127.

[105] Malte Mues, Falk Howar, and Simon Dierl. “SPouT: Symbolic Path Recording during Testing - a Concolic Executor for the JVM.” In: *20th International Conference on Software Engineering and Formal Methods*. Springer, 2022, pp. 91–107. DOI: 10.1007/978-3-031-17108-6_6.

[106] Malte Mues, Till Schallau, and Falk Howar. *Artifact for 'Jaint: A Framework for User-Defined Dynamic Taint-Analyses based on Dynamic Symbolic Execution of Java Programs'.* Version v1. Sept. 2020. DOI: `10.5281/zenodo.4060244`. URL: `https://doi.org/10.5281/zenodo.4060244`.

[107] Malte Mues, Till Schallau, and Falk Howar. "Jaint: A Framework for User-Defined Dynamic Taint-Analyses Based on Dynamic Symbolic Execution of Java Programs." In: *Integrated Formal Methods.* Ed. by Brijesh Dongol and Elena Troubitsyna. Cham: Springer International Publishing, 2020, pp. 123–140. DOI: `10.1007/978-3-030-63461-2_7`.

[108] Håvard Myrbakken and Ricardo Colomo-Palacios. "DevSecOps: A Multivocal Literature Review." In: *Software Process Improvement and Capability Determination.* Ed. by Antonia Mas et al. Cham: Springer International Publishing, 2017, pp. 17–29. DOI: `10.1007/978-3-319-67383-7_2`.

[109] Abbas Naderi-Afooshteh et al. "Joza: Hybrid Taint Inference for Defeating Web Application SQL Injection Attacks." In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE. 2015, pp. 172–183. DOI: `10.1109/DSN.2015.13`.

[110] Greg Nelson and Derek C Oppen. "Simplification by cooperating decision procedures." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.2 (1979), pp. 245–257. DOI: `10.1145/357073.357079`.

[111] Anh Nguyen-Tuong et al. "Automatically Hardening Web Applications Using Precise Tainting." In: *Security and Privacy in the Age of Ubiquitous Computing.* Ed. by Ryoichi Sasaki et al. Boston, MA: Springer, 2005, pp. 295–307. DOI: `10.1007/0-387-25660-1_20`.

[112] Aina Niemetz and Mathias Preiner. "Bitwuzla at the SMT-COMP 2020." In: *CoRR* abs/2006.01621 (2020). arXiv: `2006.01621`. URL: `https://arxiv.org/abs/2006.01621`.

[113] Aina Niemetz and Mathias Preiner. "Ternary Propagation-Based Local Search for more Bit-Precise Reasoning." In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020.* IEEE, 2020, pp. 214–224. DOI: `10.34727/2020/isbn.978-3-85448-042-6_29`.

[114] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0." In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: `10.3233/sat190101`.

[115] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)." In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[116] Carlos Pacheco et al. "Feedback-directed random test generation." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 75–84. DOI: `10.1109/ICSE.2007.37`.

[117] Hristina Palikareva and Cristian Cadar. "Multi-solver Support in Symbolic Execution." In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 53–68. DOI: `10.1007/978-3-642-39799-8_3`.

[118] Corina S Păsăreanu and Neha Rungta. "Symbolic PathFinder: Symbolic Execution of Java Bytecode." In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 179–180. DOI: `10.1145/1858996.1859035`.

[119] Corina S Păsăreanu et al. "Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis." In: *Advances in Computers*. Vol. 113. Elsevier, 2019, pp. 289–314. DOI: `10.1016/bs.adcom.2018.10.004`.

[120] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. "Symbolic execution of programs with strings." In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. 2012, pp. 139–148. DOI: `10.1145/2389836.2389853`.

[121] Andrew Reynolds et al. "Scaling up DPLL(T) String Solvers Using Context-Dependent Simplification." In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 453–474. DOI: `10.1007/978-3-319-63390-9_24`.

[122] John R Rice. "The Algorithm Selection Problem." In: *Advances in computers*. Vol. 15. Elsevier, 1976, pp. 65–118. DOI: `10.1016/S0065-2458(08)60520-3`.

[123] Heinz Riener et al. "metaSMT: focus on your application and not on solver integration." In: *International Journal on Software Tools for Technology Transfer* 19.5 (2017), pp. 605–621. DOI: `10.1007/s10009-016-0426-1`.

[124] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security." In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19. DOI: `10.1109/JSAC.2002.806121`.

[125] P. Saxena et al. "A Symbolic Execution Framework for JavaScript." In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 513–528. DOI: `10.1109/SP.2010.38`.

[126] Daniel Schoepe et al. "Explicit Secrecy: A Policy for Taint Tracking." In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 15–30. DOI: `10.1109/EuroSP.2016.14`.

[127] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask).” In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[128] Joseph Scott, Federico Mora, and Vijay Ganesh. “BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers.” In: *Software Verification*. Ed. by Maria Christakis et al. Cham: Springer International Publishing, 2020, pp. 68–86. DOI: 10.1007/978-3-030-63618-0_5.

[129] Roberto Sebastiani and Patrick Trentin. “OptiMathSAT: A Tool for Optimization Modulo Theories.” In: *Proc. International Conference on Computer-Aided Verification, CAV 2015*. Vol. 9206. LNCS. Springer, 2015. DOI: 978-3-319-21690-4_27.

[130] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C.” In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750.

[131] Ali Shamakhi, Hossein Hojjat, and Philipp Rümmer. “Towards String Support in JayHorn (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 443–447. DOI: 10.1007/978-3-030-72013-1_29.

[132] Daryl Shannon et al. “Abstracting Symbolic Execution with String Analysis.” In: *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE. 2007, pp. 13–22. DOI: 10.1109/TAIC.PART.2007.34.

[133] Vaibhav Sharma et al. “Java Ranger at SV-COMP 2020 (competition contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2020, pp. 393–397. DOI: 10.1007/978-3-030-45237-7_27.

[134] Asia Slowinska and Herbert Bos. “Pointless tainting?: evaluating the practicality of pointer tainting.” In: *Proceedings of the 4th ACM European conference on Computer systems*. ACM. 2009, pp. 61–74. DOI: 10.1145/1519065.1519073.

[135] Dawn Song et al. “BitBlaze: A New Approach to Computer Security via Binary Analysis.” In: *Information Systems Security*. Ed. by R. Sekar and Arun K. Pujari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25. DOI: 10.1007/978-3-540-89862-7_1.

[136] Fausto Spoto. "The Julia Static Analyzer for Java." In: *Static Analysis*. Ed. by Xavier Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 39–57. DOI: 10.1007/978-3-662-53413-7_3.

[137] Sanu Subramanian et al. "A Solver for a Theory of Strings and Bit-Vectors." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 124–126. DOI: 10.1109/ICSE-C.2017.73.

[138] Nikolai Tillmann and Jonathan De Halleux. "Pex–White Box Test Generation for .NET." In: *International conference on tests and proofs*. Springer. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10.

[139] Omer Tripp et al. "TAJ: effective taint analysis of web applications." In: *ACM Sigplan Notices* 44.6 (2009), pp. 87–97. DOI: 10.1145/1543135.1542486.

[140] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus." In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[141] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. "Green: reducing, reusing and recycling constraints in program analysis." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11. DOI: 10.1145/2393596.2393665.

[142] Willem Visser et al. "Model checking programs." In: *Automated software engineering* 10.2 (2003), pp. 203–232. DOI: 10.1023/A:1022920129859.

[143] Dave Wichers and Jeff Williams. "Owasp top-10 2017." In: *OWASP Foundation* (2017).

[144] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 718–730. DOI: 10.1145/3385412.3385985.

[145] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. "A concurrent portfolio approach to SMT solving." In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 715–720. DOI: 10.1007/978-3-642-02658-4_60.

[146] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. "Efficiently solving quantified bit-vector formulas." In: *Formal Methods Syst. Des.* 42.1 (2013), pp. 3–23. DOI: 10.1007/s10703-012-0156-2.

[147] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. “Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks.” In: *USENIX Security Symposium*. 2006, pp. 121–136.

Eidesstattliche Versicherung

Hiermit versichere ich, Malte Paul Mues, dass die Dissertation von mir selbstständig angefertigt wurde und alle von mir genutzten Hilfsmittel angegeben wurden. Ich versichere, dass alle in Anspruch genommenen Quellen und Hilfen in der Dissertation vermerkt wurden.

Ort, Datum

Unterschrift