Software-Engineering Seminar, Winter 2017/18

LaTeX Tutorial

Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

LaTeX

You write your document in plain text with commands that describe its structure and meaning.

The LaTeX program processes your text and produces PDF.

Idea: Focus on content, let LaTeX do the layout.

LaTeX

You write your document in plain text with commands that describe its structure and meaning.

The LaTeX program processes your text and produces PDF.

Idea: Focus on content, let LaTeX do the layout.

- Use provided style
- Avoid manual layout adjustments
- Avoid manual line and page breaks

Compiler and editors

- TeXStudio
- Kile
- TeXlipse
- Emacs
- Atom
- ...

Demo

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Compiler and editors

- TeXStudio
- Kile
- TeXlipse
- Emacs
- Atom
- ...

Demo

- Compile often, errors not always useful, focus on first error
- Use synctex to jump from PDF to source
- Configure a spellchecker for your editor
- Online editors like Overleaf or Sharelatex not recommended

Text, newlines, and paragraphs

LaTeX	PDF

`Linebreaks`
`and additional     spaces    are`
`ignored in the output.`
 	
`Empty lines separate paragraphs.`
 	
`Manual linebreaks \\`
`are possible, but`
`should be avoided.`
	Linebreaks and additional spaces are ignored in the output.
	
	Empty lines separate paragraphs.
	
	Manual linebreaks
	are possible, but should be avoided.

Special symbols

LaTeX	PDF

`Double ‘‘Quotes’’ and single ‘quotes’.`
 	
`Wrong ”quotes”.`
 	
`% a comment`
	Double “Quotes” and single ‘quotes’.
	
	Wrong ”quotes”.

Commands

LaTeX	PDF
`Commands start with a backslash, for example: \textbf bold font.`	Commands start with a backslash, for example: **b**old font.
`Curly braces group text, for example: \textbf{bold font}.`	Curly braces group text, for example: **bold font**.
`Square brackets for optional arguments, as in \lstinline[language=Java]{if (x <3) throw new Exception()}`	Square brackets for optional arguments, as in `if (x<3) throw new Exception()`

Other special symbols

LaTeX	PDF
`Special symbols can be escaped with a backslash.`	Special symbols can be escaped with a backslash.
`For example: \$ \% \& \# \_`	For example: $ % & # _

Document structure

-1	`\part{...}`
0	`\chapter{...}`
1	`\section{...}`
2	`\subsection{...}`
3	`\subsubsection{...}`
4	`\paragraph{...}`
5	`\subparagraph{...}`

`\section`, `\subsection` and `\paragraph` usually enough for papers.

`\part` and `\chapter` are only available in report and book document classes.

Add a * to remove numbers, e.g. `\section*{...}`

Lists

LaTeX	PDF

`\begin{itemize}`
` \item Unordered`
` \item List`
` \item \dots`
`\end{itemize}`
	■ Unordered
	■ List
	■ ...

`\begin{enumerate}`
` \item Numbered`
` \item list`
` \item \dots`
`\end{enumerate}`
	1 Numbered
	2 list
	3 ...

Definition lists

LaTeX	PDF

`\begin{description}`
`\item[Word A] Word A is \dots`
`\item[Word B] B is \dots`
`\end{description}`
	Word A Word A is . . .
	Word B B is . . .

Tables

```
\begin{tabular}{lcr}
Place     & Food        & Price \\
Ausgabe 1 & Rahmbraten  & 2.40 \\
Ausgabe 2 & Tagliatelle & 2.15 \\
Atrium    & Kebab       & 3.90 \\
\end{tabular}
```

Place	Food	Price
Ausgabe 1	Rahmbraten	2.40€
Ausgabe 2	Tagliatelle	2.15€
Atrium	Kebab	3.90€

Tables

```
\begin{tabular}{|l|c|r|}
Place     & Food        & Price \\ \hline
Ausgabe 1 & Rahmbraten  & 2.40 \\
Ausgabe 2 & Tagliatelle & 2.15 \\
Atrium    & Kebab       & 3.90 \\
\end{tabular}
```

Place	Food	Price
Ausgabe 1	Rahmbraten	2.40€
Ausgabe 2	Tagliatelle	2.15€
Atrium	Kebab	3.90€

Code Listings

```
\begin{lstlisting}
public static void main(String[] args) {
  // some comment
  System.out.println("Hello World!");
}
\end{lstlisting}
```

```
public static void main(String[] args) {
  // some comment
  System.out.println("Hello World!");
}
```

Code Listings

```
\begin{lstlisting}[language=Java]
public static void main(String[] args) {
  // some comment
  System.out.println("Hello World!");
}
\end{lstlisting}
```

```java
public static void main(String[] args) {
  // some comment
  System.out.println("Hello␣World!");
}
```

Code Listings

```
\begin{lstlisting}[language=Java,morekeywords={out,println}, numbers=
left]
public static void main(String[] args) {
  // some comment
  System.out.println("Hello World!");
}
\end{lstlisting}
```

```java
public static void main(String[] args) {
  // some comment
  System.out.println("Hello␣World!");
}
```

Figures

```java
public static void main(String[] args) {
  // some comment
  System.out.println("Hello␣World!");
}
```

Figure 1: A simple Java program

```latex
\begin{figure}
\begin{lstlisting}[language=Java]
public static void main(String[] args) {
  // some comment
  System.out.println("Hello World!");
}
\end{lstlisting}
\caption{A simple Java program}
\label{fig:java_example}
\end{figure}
```

Labels and References

```
Use the label name to reference Figure \ref{fig:java_example}.
```

Use the label name to reference Figure 1.

Labels and References

```
Use the label name to reference Figure \ref{fig:java_example}.
```

Use the label name to reference Figure 1.

Labels can also be used to reference sections:

```
\section{Part1}
\label{sec:part1}

\subsection{Details}
\label{sec:part1a}
```

Images

`\includegraphics[width=10cm]{bitcoin.png}`

Source: Ladislav Mecir, https://en.wikipedia.org/wiki/File:Bitcoin_price_and_volatility.svg

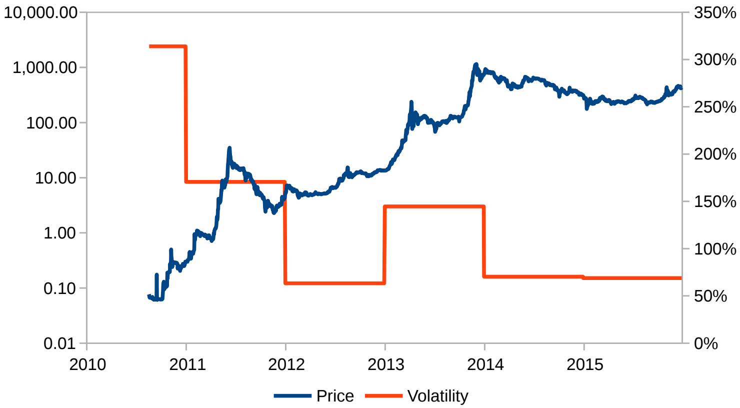
Images

```
\includegraphics[width=10cm]{bitcoin_hd.png}
```

Source: Ladislav Mecir, https://en.wikipedia.org/wiki/File:Bitcoin_price_and_volatility.svg

Images

```
\includegraphics[width=10cm]{bitcoin.pdf}
```



Source: Ladislav Mecir, https://en.wikipedia.org/wiki/File:Bitcoin_price_and_volatility.svg

Images

Use images in Figures.

Use vector images (pdf) instead of rasterized images (png, jpg) if possible.

Use your own graphics if possible, otherwise reference source.

Formulas

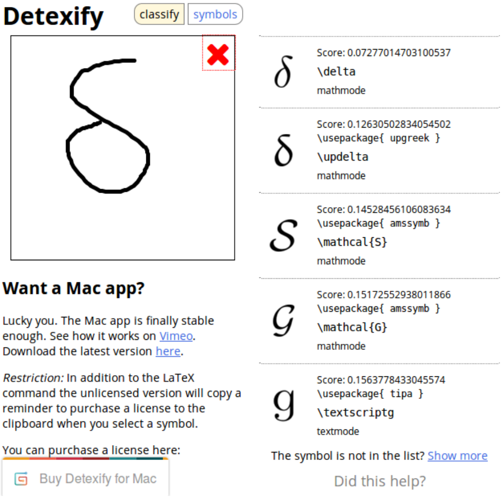
```
Formulas can be used inline
$\sum_{i=1}^\infty {6 \over i^2} = \pi^2$
or in a block:

\[ \sum_{i=1}^\infty {6 \over i^2} = \pi^2 \]
```

Math formulas can be used inline $\sum_{i=1}^\infty \frac{6}{i^2} = \pi^2$ or in a block:

$$\sum_{i=1}^\infty \frac{6}{i^2} = \pi^2$$

Formulas

Use detexify (http://detexify.kirelabs.org/) to find Latex symbols.



Citations

Add Bibtex entry to `references.bib`:

```
@inproceedings{dobedobedo,
  author    = {Sam Lindley and
               Conor McBride and
               Craig McLaughlin},
  title     = {Do be do be do},
  booktitle = {Proceedings of the 44th {ACM} {SIGPLAN} Symposium on
               Principles of Programming Languages,
               {POPL} 2017, Paris, France, January 18-20,
               2017},
  year      = {2017},
  url       = {http://dl.acm.org/citation.cfm?id=3009897},
}
```

Reference in Text:

```
Frank \cite{dobedobedo} is a language with effect handlers but no
separate notion of function: a function is but a special case of a
handler.
```

Citing online resources

```
@misc{discord,
  title = {How Discord Stores Billions of Messages},
  author = {Stanislav Vishnevskiy},
  howpublished = {\url{https://blog.discordapp.com/how-discord-stores-
billions-of-messages-7fa6ec7ee4c7}},
  note = {Accessed: 2017-10-12}
}
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Structure

and discuss our experiences reporting these vulnerabilities to developers, who have confirmed several thus far. We evaluate which databases provide sufficiently strong isolation guarantees to prevent these attacks. Of the 22 vulnerabilities, 17 occur due to incorrect transaction usage and are therefore not preventable without substantial code modification. We investigate common program behavior among vulnerable and non-vulnerable code paths and present constructive strategies for preventing attacks.

The remainder of this paper proceeds as follows. Section 2 defines ACIDRain attacks. In Section 3, we develop and formally motivate the 2AD analysis theory. Section 4 describes our experiences detecting and exploiting real vulnerabilities in eCommerce applications. Section 5 discusses related work, and Section 6 concludes.

2. ACIDRain ATTACKS

In this section, we define ACIDRain attacks more precisely and

Structure

and relational operators. A *contract enforcement system* statically maps operations over the datatype to a particular consistency level available on the store, and provably validates the correctness of the mapping. The paper makes the following contributions:

- We introduce QUELEA, a shallow extension of Haskell that supports the description and validation of replicated data types found in an ECDS. Contracts are used to specify fine-grained application-level consistency properties, and are statically analyzed to assign the most efficient and sound store consistency level to the corresponding operation.
- QUELEA supports coordination-free transactions over arbitrary datatypes. We extend our contract language to express fine-grained transaction isolation guarantees, and utilize the contract enforcement system to automatically assign the correct isolation level for a transaction.

Structure

The rest of the paper is organized as follows. The next section describes the system model. We describe the challenges in programming under eventual consistency, and introduce QUELEA contracts as a proposed solution to overcome these issues in § 3. § 4 provides more details on the contract language, and its mapping to store consistency levels, along with meta-theory for certifying the correctness of the mapping. § 5 introduces transaction contracts and their classification. § 6 describes the implementation of QUELEA on top of Cassandra. § 7 discusses experimental evaluation. § 8 and 9 present related work and conclusions.

2. System Model

In this section, we describe the system model and introduce the primitive relations that our contract language is seeded with. Figure 1 presents a schematic diagram of our system model. The distributed

Sentence and Paragraph length

serializable behavior during concurrent API calls. That is, while the gold standard of transaction isolation (serializable isolation) guarantees equivalence to some serial execution of transactions, not all databases will enforce serializability. Some databases do not provide serializability as an option at all, while others allow applications to select a weaker isolation mode [17, 19]. Under weaker isolation levels, transactions are subject to an array of behaviors that cannot occur under serial execution, the exact set of which depends on the particular isolation level and database [17]. We call these conventional isolation anomalies *level-based isolation anomalies* as they arise due to the database executing under non-serializable isolation levels.

Second, independent of the isolation level used, the transaction programming model requires the application to correctly encapsulate its logic within transactions. In the absence of explicit `BEGIN TRANSACTION` and `COMMIT/ABORT` commands, by default, many databases such as MySQL and PostgreSQL automatically execute each SQL operation as a separate transaction. As a result, if a web application performs multiple database operations without using transactions while servicing a single API request, then concurrent API requests may result in behavior that could not have arisen during a serial execution of API calls. We call these isolation anomalies arising from a lack of transactional encapsulation *scope-based isolation anomalies*. In this paper, we consider scoping at the level of individual API calls.

Given a set of isolation anomalies, we must determine whether any of these anomalies result in significant application behavior:

C2: Sensitive invariants. The anomalies arising from concurrent access lead to violations of application *invariants*.

In general, per Kung and Papadimitriou [45], every anomaly is problematic for *some* application; however, for a *given* application, is a given anomaly problematic? Again borrowing from the classical transaction processing literature, we capture key application properties via *invariants*, or logical predicates capturing an application's consistency criteria [34]. For example, an application might have an invariant that user IDs within a database are unique. Another application might specify that total revenue equals the sum of total orders placed. Each invariant is susceptible to violation under a particular set of anomalies.

cally identifies potential isolation anomalies. Determining invariants is more complicated, requiring either user interaction, invariant mining, or program analysis [32, 33]. As a result, in this paper we focus on a specific, concrete set of invariants found in eCommerce applications and examine a set of popular eCommerce applications to determine their susceptibility to attacks on these key invariants.

Threat model. We assume that an attacker can only access the web application via concurrent requests against publicly-accessible APIs (e.g., HTTP, REST). That is, to perform an ACIDRain attack, the attacker does not require access to the application server, database server, execution environment, or logs. Our proposed analysis techniques (Section 3) use full knowledge of the database schema and SQL logs, but, once identified, an attacker can exploit the vulnerabilities we consider here using only programmatic APIs.[2] This threat model applies to most Internet sites today.

3. 2AD: DETECTING ANOMALIES

ACIDRain attacks stem from anomalies that occur during concurrent execution. Detecting these anomalies is challenging. Many potential anomalies are never triggered under normal operation due to limited concurrency, rendering simple observation ineffective. We could use static analysis tools [50] to analyze an application's susceptibility to attacks. However, web applications are written using a variety of frameworks and languages. As a result, static analysis tools would necessarily have limited applicability.

To address these challenges, we developed a new, cross-platform methodology for detecting potential level-based and scope-based anomalies in web applications by analyzing logs of typical database activity. We call this approach *Abstract Anomaly Detection (2AD)*. Figure 2 shows an overview of the 2AD workflow.

Overview. The core idea behind 2AD is to execute API calls against a live application and database to generate a (possibly sequential) trace of database activity, then analyze the trace for potential anomalies that could arise under concurrent execution. This approach leverages the facts that our target applications all *i.*) expose API endpoints (e.g., via HTTP) that can be triggered programmati-

[2]That is, to efficiently identify vulnerabilities, our analysis makes use of non-public information in the form of database logs (e.g. SQL traces) and database schemas. However, the vulnerabilities themselves can be exploited

Linking sections

more fine-grained analysis and is a worthwhile area for future work. However, despite its limitations, 2AD has proven a useful tool in analyzing real applications—the subject of the next section.

4. ACIDRain IN THE WILD

Having described how to use database traces to identify possible anomalies, in this section we describe how to use these this approach to detect vulnerabilities and subsequently perform ACIDRain attacks. We apply a prototype 2AD analysis tool to a suite of 12 eCommerce applications, identifying 22 new ACIDRain attacks. Section 4.1 describes how to produce vulnerabilities from anomalies, and Section 4.2 details our experience finding vulnerabilities in self-hosted eCommerce applications.

4.1 From Anomalies to Vulnerabilities

Use examples

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and

(a)

```
def withdraw(amt, user_id):
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
```

(b)

```
def withdraw(amt, user_id):
  beginTxn()
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
  commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the `withdraw` function could both read balance $\$100$, check that $\$100 \geq \$99$, and each allow $\$99$ to be withdrawn, resulting $\$198$ total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as `SELECT FOR UPDATE` is used. While this scenario closely re-