Off the Beaten Path: Machine Learning for Offensive Security

Konrad Rieck
University of Göttingen
6th ACM Workshop AISEC
About me

» Konrad Rieck
  » Fun with security and machine learning for 10 years
  » Research group at the University of Göttingen

» Research focus: intelligent security systems
  » Intrusion detection, malware analysis, vulnerability discovery
Application of learning algorithms to security problems

- A long series of research in both fields
- Some amazing papers & several not so amazing papers
- **Main goal:** reduction of manual work in security

**... here’s a small study**

![USENIX Security Symposium](chart.png)

20-30% papers contain machine learning terms
Security and Learning

» Defensive security main application area for learning
  » Detection of threats: malware, intrusions, spam, ...
  » Analysis of threats: malware, spam, fraud, phishing, ...

Context of machine learning terms in papers

Why should we care about offensive security?
Defensive and Offensive Security

Defensive security

- Detection of attacks
- Analysis of attacks
- Prevention of attacks
Defensive and Offensive Security

Defensive security
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

Offensive security
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities
Defensive and Offensive Security

Defensive security:
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

Offensive security:
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities
Defensive and Offensive Security

Defensive security
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

Offensive security
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities
Offensive Security and Learning

» Why apply machine learning for offensive security?

» Many problems involve tedious manual analysis
  » Auditing of program code, analysis of monitored data
  » Manual identification of relevant patterns time-consuming

» Errors less costly in comparison to defensive security
  » False positives and false negatives less severe
  » Analysis assisted by machine learning – not replaced
Why not to apply machine learning for offensive security?

Application not necessary straightforward

- Exiting defenses usually better than expected
- Consideration of ethical issues (and sci-fi visions)
Where does learning fit?

**Offensive security**

- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities

**Machine learning**

- Classification
- Anomaly detection
- Component analysis
- Generative models
- Structured learning?
- Reinforcement learning?
Current Research: Fun with Data

» Finding data leaks and side channels
  » Identification of patterns in complex and noisy data

» Examples
  » Acoustic side-channel attacks on printers (Backes et al., USENIX SEC 2010)
  » Reconstruction of encrypted VoIP conversations (White et al., S&P 2011)
  » De-pseudonymization of smart metering (Jawurek et al., ACSAC 2011)
  » Website fingerprinting in Tor traffic (Cai et al., CCS 2012)
  » … other interesting approaches

Case study 1
Current Research: Fun with Code

» Finding vulnerabilities in program code
  » Identification of patterns indicating vulnerable code

» Examples
  » Localizing bugs in program execution (Dietz et al., NIPS 2009)
  » Vulnerability extrapolation (Yamaguchi et al., ACSAC 2012)
  » Identification of unpatched code clones (Jang et al., S&P 2012)
  » Detection of missing security checks (Yamaguchi et al., CCS 2013)
  » … other interesting approaches
Smart Metering De-Pseudonymization

Marek Jawurek, Martin Johns, and Konrad Rieck

ACSAC 2011
» **Smart metering**
  » Digital metering of electricity, water and gas consumption
  » Fine-grained recording and transmission of metering data

» **Advantages**
  » Flexible billing models
  » Optimization of capacities
Privacy and Smart Metering

» Private information observable in electricity consumption
  » Numerous approaches for identifying activities and devices
  » Data shared by several parties (suppliers, distributors, …)

» Cool example: Recognition of movie scenes

Learning of brightness profiles per scene

Identification of movie scenes

Correlation with electricity consumption

(Graveler et al., Sicherheit 2012)
Pseudonymization

» Proposed solution: pseudonymization of data
  » Storage of consumption traces with pseudonyms
  » Existing privacy attacks seemingly ineffective
  » Possibility to share data for analysis and optimization

Identity database

| John Smith      | θd j32d27 |
| Dr. Benway     | x73hwd9   |
| Frank Booth    | k43x0s24  |

Pseudonymized consumption traces

| x73hwd9   |
| k43x0s24  |
| θd j32d27 |
De-Pseudonymization

» Idea: Put security of pseudonymization to test
  » Application of machine learning to identify households

» Unsupervised and unsupervised strategies
  » Linking by behavioral patterns (Classification)
  » Linking by behavioral anomalies (Anomaly detection)
Feature Extraction

» Uncover patterns in electricity consumption
  » Influence of household’s behavior and appliances
  » Observable information: *time* and *amount* of consumption

» *Surprising finding: stateful usage of electricity*

Grid plot for 24 hours

Average consumption of a household
Stateful Feature Space

- Map from recorded consumption trace to feature space
- Consumption $x$ is sequence of $d$ measurements $m_i$
  $$x = (m_1, m_2, \ldots, m_d)$$
- Space spanned by grid of states $g$ and resolution $d$
  $$\phi : X \mapsto \mathbb{R}^{d \cdot g}, \quad \phi(x) = (l_{j,k}(x))_{1 \leq j \leq d}^{1 \leq k \leq g}$$
- with indicator $l_{i,k}(x) = \begin{cases} 1 & \text{if } m_i \text{ falls into state } k \\ 0 & \text{otherwise.} \end{cases}$
Learning for Linking

» Linking by behavioral patterns
  » Classification of households’ consumption traces
  » Linking of different traces of the same household
  » *Learning method:* linear two-class SVM

» Linking by anomalies
  » Anomaly detection on households’ consumption traces
  » Linking of detected anomalies with external observations
  » *Learning method:* simple mean vector ;)

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN
Linking by Behavioral Pattern

- **Study with 53 households**
  - Recording of 24 values per day over 7 months
  - Map to stateful feature spacing with 100 state bins
  - **Accuracy of de-pseudonymization: 82%**

![Per-consumer accuracy](chart1.png)

![Influence of data sizes](chart2.png)
Linking by Anomalies

Detecting of anomalies

» Often plausible events

» No ground truth available
Consequences

» Pseudonymization insufficient to protect privacy

» Other weak strategies
  » Frequent re-pseudonymization of collected data
  » Lower metering resolution

» Recommended solution
  » On-site computation of tariffs (e.g. in the meter)
Vulnerabilities in Code

» Discovery of vulnerabilities far from trivial
  » Some low-hanging fruits, e.g. `strcat`, `strcpy`, `sprintf`, ...
  » More often subtle errors in programming

» Current strategies for discovery of vulnerabilities in code
  » Testing and fuzzing of implementations
  » Taint analysis and symbolic execution
  » ... still many bugs only discovered by manual auditing

¶ A universal discovery method cannot exist (Rice’s theorem)
A “Modern” Vulnerability

```c
static int receiveauthgrant(OscarData *od,
    FlapConnection *conn,
    aim_module_t *mod,
    FlapFrame *frame,
    aim_modsnac_t *snac,
    ByteStream *bs)
{
    int ret = 0;
    aim_rxcallback_t userfunc;
    guint16 tmp;
    char *bn, *msg;
    /* Read buddy name */
    if ((tmp = byte_stream_get8(bs)))
        bn = byte_stream_getstr(bs, tmp);
    else
        bn = NULL;
    /* Read message (null terminated) */
    if ((tmp = byte_stream_get16(bs)))
        msg = byte_stream_getstr(bs, tmp);
    else
        msg = NULL;
    /* Unknown */
    tmp = byte_stream_get16(bs);
    if ((userfunc =
        aim_callhandler(od, snac->family, snac->subtype)))
        ret = userfunc(od, conn, frame, bn, msg);
    g_free(bn);
    g_free(msg);
    return ret;
}
```

Vulnerability in Pidgin (CVE-2011-4601)

- Read buddy name (usually UTF-8)
- Read message (usually UTF-8)
- Call message handler (UTF-8 required)
- Missing input validation → Segfault
Vulnerability Extrapolation

» **Motivation from practice**
  » *Scenario:* a vulnerability has just been reported.
  » Do other similar vulnerabilities exist in the code?
  » Is there a problematic programming pattern?

» **Idea: Extrapolation of vulnerabilities**
  » Machine learning for finding similar vulnerabilities
  » Modeling and extraction of programming patterns
» Extraction of abstract syntax trees (ASTs)
  » Robust parsing using island grammar
  » Representation of each function as AST
  » Focus on usage of APIs and syntax

Example function

```c
int foo(int y) {
    int n = bar(y);
    if (n == 0)
        return 1;
    return n + y;
}
```

API node

Syntax node
Subtrees in ASTs

» Programming patterns reflected in subtrees

» Three types of subtrees considered

---

API nodes \((d=1)\)

- **What APIs are used?**
  - call:x
  - int

API subtrees \((d>1)\)

- **How are APIs used?**
  - call:x
  - int
  - int
  - call:x

API/S subtrees \((d>1)\)

- **How are APIs and syntax used?**
  - call:x
  - int
  - +
  - int
  - call:x
  - +
From Trees To Vectors

- **Embedding of ASTs in vector space**
  - Characterization of trees using set of subtrees $S$
  - Each subtree $s \in S$ associated with one dimension
  - *Dimension* = frequency $\#(s,x)$ of subtree $s$ in tree $x$

$$\phi : X \rightarrow \mathbb{R}^{|S|}, \quad \phi(x) \mapsto \left( \#(s,x) \cdot w_s \right)_{s \in S}$$

- **ASTs mapped to vector space. Now what?**
  - Application of many learning methods possible
Latent Semantic Analysis

- Standard method from the area of text mining
- Automatic identification of topics in text documents
- *Topic vector* = frequent combination of words

\[
\begin{pmatrix}
0.75 \\
\ldots \\
0.90 \\
\ldots \\
0.10 \\
\ldots \\
0.20
\end{pmatrix}
\]

Documents on fruit salad

banana
apple
food
“Topics” in Source Code?

» Latent semantic analysis on embedded trees
  » Automatic identification of programming patterns in code
  » *Programming pattern* = frequent combination of subtrees

```
foo
```
```
call: bar
```

Pattern involving “foo” and “bar”
Extrapolation

» **Projection of code**
  » ASTs projected to space of programming patterns
  » Dimension reduction and denoising of code

» **Extrapolation of vulnerability**
  » Mapping of known vulnerability to projected space
  » Nearest-neighbor classification with one training point ;)

Known vulnerability
Candidates
» Does this “extrapolation” really work?

» Four open-source projects with a known vulnerability
  » LibTIFF: image processing library with 1,292 functions
  » FFmpeg: video streaming library with 6,941 functions
  » Pidgin: chat client with 11,505 functions
  » Asterisk: VoIP server with 8,155 functions

» Ground truth: manually labeled candidate functions
  » Two weeks of work for an excellent PhD student
Quantitative Evaluation

» How much code do we need to audit?
  » Functions sorted according to similarity with vulnerability
  » *Criteria:* Code to audit before seeing all candidates

<table>
<thead>
<tr>
<th>API nodes</th>
<th>API subtrees</th>
<th>API/S subtrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>LibTIFF</td>
<td>7 %</td>
<td>17 %</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>20 %</td>
<td>28 %</td>
</tr>
<tr>
<td>Pidgin</td>
<td>2 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Asterisk</td>
<td>15 %</td>
<td>28 %</td>
</tr>
</tbody>
</table>

» All candidates discovered in first 15% of ranked functions
» Assistance during auditing: speed-up factor of ~11x
The discovery of vulnerable code in software is a hard problem. Due to the fundamental inability of one program to completely analyze another program’s code, a generic technique for finding arbitrary vulnerabilities does not exist. As a consequence, all practical approaches either potentially identify vulnerable code in Pidgin.

Table 3: Top 30 most similar functions to a known vulnerability.

```
<table>
<thead>
<tr>
<th>Sim. Function name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>receiveauthgrant</td>
<td>0.99</td>
</tr>
<tr>
<td>parseicon</td>
<td>0.98</td>
</tr>
<tr>
<td>parsepopup</td>
<td>1.00</td>
</tr>
<tr>
<td>parseinfo_create</td>
<td>1.00</td>
</tr>
<tr>
<td>parserights</td>
<td>1.00</td>
</tr>
<tr>
<td>incomingim_ch2_sendfile</td>
<td>1.00</td>
</tr>
<tr>
<td>uploadack</td>
<td>1.00</td>
</tr>
<tr>
<td>incomingim</td>
<td>1.00</td>
</tr>
<tr>
<td>generror</td>
<td>0.99</td>
</tr>
<tr>
<td>incoming_.._buddylist</td>
<td>0.99</td>
</tr>
<tr>
<td>parsemod</td>
<td>0.99</td>
</tr>
<tr>
<td>userinfo</td>
<td>0.99</td>
</tr>
<tr>
<td>motd</td>
<td>0.99</td>
</tr>
<tr>
<td>receiveadded</td>
<td>0.99</td>
</tr>
</tbody>
</table>
```

The limitation of our approach in more detail.

Figure 7: Original vulnerability (CVE-2011-4601) in Pidgin.

```c
static int
receiveauthgrant(OscarData *od,
    FlapConnection *conn,
    aim_module_t *mod,
    FlapFrame *frame,
    aim_modsnac_t *snac,
    ByteStream *bs)
{
    int ret = 0;
    aim_rxcallback_t userfunc;
    guint16 tmp;
    char *bn, *msg;
    /* Read buddy name */
    if ((tmp = byte_stream_get8(bs)))
        bn = byte_stream_getstr(bs, tmp);
    else
        bn = NULL;
    /* Read message (null terminated) */
    if ((tmp = byte_stream_get16(bs)))
        msg = byte_stream_getstr(bs, tmp);
    else
        msg = NULL;
    /* Unknown */
    if ((userfunc =
        aim_callhandler(od, snac->family, snac->subtype)))
        ret = userfunc(od, conn, frame, bn, iconcsumtype, iconcsumlen, iconcsum, *icon);
    bn = byte_stream_getstr(bs, byte_stream_get8(bs));
    iconcsumtype = byte_stream_get8(bs);
    iconcsumlen = byte_stream_get8(bs);
    iconcsum = byte_stream_getraw(bs, iconcsumlen);
    iconlen = byte_stream_get16(bs);
    icon = byte_stream_getraw(bs, iconlen);
    bn = byte_stream_getstr(bs, byte_stream_get16(bs));
    if ((userfunc =
        aim_callhandler(od, snac->family, snac->subtype)))
        ret = userfunc(od, conn, frame, bn, iconcsumtype, iconcsumlen, icon, iconlen);
    g_free(bn);
    g_free(icon);
    return ret;
}
```
The discovery of vulnerable code in software is a hard problem. Due to the fundamental inability of one program to completely analyze another program's code, a generic technique for finding arbitrary vulnerabilities does not exist. Moreover, related techniques such as fuzz testing, taint analysis, and symbolic execution can be easily coupled with vulnerability analysis. These techniques are effective in identifying vulnerabilities, but they do not provide any guarantee that the discovered vulnerabilities are present in the program under analysis. As a consequence, all practical approaches either limit the search to specific types of vulnerabilities or, as in the case of vulnerability extrapolation, only identify potentially vulnerable code, yet they do not provide any guarantee that the discovered vulnerabilities are present in the program under analysis. This limitation is inherent to the application of techniques for finding and eliminating security flaws in source code rather than the true semantics. Due to Rice's theorem, automatic vulnerability extrapolation is impossible. However, our case study shows that vulnerabilities distributed over two functions can still be effectively identified, as long as both functions share some structural patterns with the original vulnerability. In cases where no known vulnerability is available, our method cannot be applied. In practice such limitations of our approach in more detail.

### LIMITATIONS

Due to the fundamental inability of one program to completely analyze another program's code, a generic technique for finding arbitrary vulnerabilities does not exist. Moreover, related techniques such as fuzz testing, taint analysis, and symbolic execution can be easily coupled with vulnerability analysis. These techniques are effective in identifying vulnerabilities, but they do not provide any guarantee that the discovered vulnerabilities are present in the program under analysis. As a consequence, all practical approaches either limit the search to specific types of vulnerabilities or, as in the case of vulnerability extrapolation, only identify potentially vulnerable code, yet they do not provide any guarantee that the discovered vulnerabilities are present in the program under analysis. This limitation is inherent to the application of techniques for finding and eliminating security flaws in source code rather than the true semantics. Due to Rice's theorem, automatic vulnerability extrapolation is impossible. However, our case study shows that vulnerabilities distributed over two functions can still be effectively identified, as long as both functions share some structural patterns with the original vulnerability. In cases where no known vulnerability is available, our method cannot be applied. In practice such
Consequences

» Extrapolation of vulnerabilities in program code possible

» Detection of unknown vulnerabilities ("0days")
  » 10 vulnerabilities discovered in the 4 open-source projects
  » Software developers informed prior to publication
  » (Hopefully) fixed in current versions

» Defensive security strengthened
  » Prevention of attacks due to vulnerability discovery
Conclusions
Conclusions

» Offensive security and machine learning
  » Fruitful combination for discovering vulnerabilities
  » Applications in the digital as well as physical world
  » Errors less problematic as in the defensive setting

» Examples: Two different case studies
  » De-pseudonymization of smart metering data
  » Extrapolation of vulnerabilities in program code
Outlook

» What could be the next steps?

» Approaches for analysis and exploitation of vulnerabilities
  » Can we generalize models from vulnerabilities?
  » Can we identify and describe vulnerable patterns?
  » Can we extend current exploitation techniques?

» Combination with adversarial learning research
  » Joint view on machine learning and security domain
Thank you! Questions?