

# ROMEO: A Binary Vulnerability Detection Dataset for Exploring Juliet through the Lens of Assembly Language

Clemens-Alexander Brust, Tim Sonnekalb, Bernd Gruner  
DLR Institute of Data Science  
Jena, Germany  
`firstname.lastname@dlr.de`

## Abstract

**Context** Automatic vulnerability detection on C/C++ source code has benefitted from the introduction of machine learning to the field, with many recent publications targeting this combination. In contrast, assembly language or machine code artifacts receive less attention, although there are compelling reasons to study them. They are more representative of what is executed, more easily incorporated in dynamic analysis, and in the case of closed-source code, there is no alternative.

**Objective** We evaluate the representative capability of assembly language compared to C/C++ source code for vulnerability detection. Furthermore, we investigate the role of call graph context in detecting function-spanning vulnerabilities. Finally, we verify whether compiling a benchmark dataset compromises an experiment’s soundness by inadvertently leaking label information.

**Method** We propose ROMEO, a publicly available, reproducible and reusable binary vulnerability detection benchmark dataset derived from the synthetic Juliet test suite. Alongside, we introduce a simple text-based assembly language representation that includes context for function-spanning vulnerability detection and semantics to detect high-level vulnerabilities. It is constructed by disassembling the `.text` segment of the respective binaries.

**Results** We evaluate an x86 assembly language representation of the compiled dataset, combined with an off-the-shelf classifier. It compares favorably to state-of-the-art methods, including those operating on the full C/C++ code. Including context information using the call graph improves detection of function-spanning vulnerabilities. There is no label information leaked during the compilation process.

**Conclusion** Performing vulnerability detection on a compiled program instead of the source code is a worthwhile tradeoff. While certain information is lost, e.g., comments and certain identifiers, other valuable information is gained, e.g., about compiler optimizations.

## 1 Introduction

Machine learning has advanced automatic vulnerability detection significantly compared to traditional static analysis security testing (SAST) tools [17]. The focus of recent methods is mainly on C/C++ source code, where many vulnerabilities can be detected reliably, but others cannot [5]. We argue that, at least for C/C++ software projects, but also other languages, the compiler’s output warrants the same attention as the source code, if not more. The machine code is more closely related to the actual state transitions during execution than the original source code, and SAST tools often fail to consider implementation details of compilers [4]. Moreover, source code is not always available for third party software, which nevertheless has to be audited. Finally, a method that works directly on assembly language or machine code can easily be transferred to a dynamic environment, *e.g.* for runtime monitoring of JIT compilers, and it supports all compiled higher-level languages by design.

While there is recent work in the area of vulnerability detection on machine code and assembly language (see section 2.1), there is no reproducible, publicly available dataset for this task. Previous work suffers from further limitations (see section 2). To the best of our knowledge, there is also no binary vulnerability method that incorporates the semantics and context information necessary to detect more abstract vulnerabilities spanning multiple functions. To advance the state of the art into this direction, we contribute the following:

**The ROMEO dataset:** a binary vulnerability detection benchmark dataset based on the Juliet test suite [10] version 1.3, with approx. 168k examples labeled as one of 91 Common Weakness Enumeration (CWE, [21]) categories. We publish the source code of the entire processing pipeline, which is reproducible and configurable to meet the needs of other researchers.

**The ROMEO representation:** a simple text representation of disassembled binaries suitable for various sequence classifiers. It incorporates context for across-function vulnerabilities and preserves semantics to identify API calls, while still preventing label leakage.

**Experiments:** using an off-the-shelf Transformer approach [9], we show the efficacy of our assembly language representation, which compares favorably even to state-of-the-art methods that have access to the full source code, and highlight the relative strengths and weaknesses in qualitative analyses.

The remainder of this work is structured as follows: we first present related work in section 2. We then give a detailed description of the dataset creation

process and our representation in section 3. Research questions are posed and answered experimentally in section 4. We finally offer a brief conclusion and an outlook towards future research in section 5.

## 2 Related Work

In this section, we review previous work related to our investigation. We focus on two main areas. First, we consider approaches to vulnerability detection that use an assembly language or machine code representation. Second, we explore work that uses the Juliet test suite as a benchmark or evaluation dataset.

### 2.1 Vulnerability Detection using Assembly Language Representations

Lee *et al.* [13] combine a bespoke encoding of assembly language instructions called Instruction2vec [12] with a deep learning model “Text-CNN” to detect vulnerabilities in binary code. The encoding represents each instruction as a vector of a fixed length. Individual components of the instruction, *i.e.* the opcode and operand fragments, are encoded using a custom word2vec [18] model. Their model detects CWE-121 (Stack Overflow) vulnerabilities in Juliet with an accuracy of 96.1% compared to an off-the-shelf word2vec at 94.2%. We replicate their setup in our evaluation for comparison.

BVDDetector [22] operates based on pre-extracted program slices. It relies on a per-token word2vec encoding. While the authors test a variety of neural networks to classify the encoded slices, they find that a BGRU [6] performs best. They measure the performance of their method on program slices extracted from a subset of the Juliet test suite concerning memory corruption and number handling vulnerabilities and report an accuracy of 96.7%. We construct a subset using the same criteria to compare our approach to BVDDetector as their dataset is not publicly available.

The notion of program slices [17, 16] is extended to assembly language by Li *et al.* [14]. They also incorporate a combined representation of source code and assembly called “hybrid slices”. The hybrid slice method is evaluated on an aggregated subset of Juliet, where it reaches 96.9% accuracy compared to BVDDetector at 88.9%.

Le *et al.* propose a Maximal Divergence Sequential Auto-Encoder [11] and rely on a fixed encoding of opcodes and a histogram-like encoding of operands to represent assembly language instructions. The autoencoder is evaluated on the dataset introduced alongside VulDeePecker [17], and outperforms it slightly with an accuracy of 85.3% vs. 83.5%.

In [1, 2], a dataset called BVATT is constructed similarly to ours. It is intended as a reliable benchmark for binary vulnerability detection methods. However, it is no longer publicly available.

All the aforementioned methods choose different learned or hand-crafted

encodings of the machine code. However, none attempt to leverage the human-readable mnemonic representation of assembly as we propose in this work.

## 2.2 Juliet Test Suite as a Benchmark for Vulnerability Detection

In the following, we provide a brief overview of related work where the Juliet test suite is used as a benchmark dataset outside its intended SAST application.

Russel *et al.* [20] augment Juliet with examples mined from GitHub repositories and Debian packages. The mined examples are annotated using the static analysis tools Clang, Cppcheck and Flawfinder. While the tool selection is diverse, it cannot provide a replacement for manually labeled data. Machine learning-based methods can simply learn the rules embedded in these tools, and thus will produce similar errors. Juliet, while synthetic, is almost by definition always labeled correctly.

Li *et al.* evaluate their method VulDeePecker [17] on a combination of CWEs 119 and 399 of the Juliet test suite and vulnerabilities in open source projects listed in the National Vulnerability Database (NVD). This dataset is compiled and used by Le *et al.* in [11] to validate their binary vulnerability detector (see above). The NVD provides compelling real-world examples, but introduces label quality concerns into the dataset due to the complex mining process involved. Furthermore, the dataset is not suitable for our evaluation because its individual examples are not compilable on their own. Li *et al.* refine and extend the combination of Juliet and examples from the NVD to evaluate SySeVR [16].

BVDetector [22] uses a subset of the Juliet test suite without any additions (see section 2.1 for further details). A similar approach, but combined with the C/C++ source code, is taken by Li *et al.* [14]. Conversely, our evaluation uses a much larger subset, missing only Windows-specific weaknesses. The Juliet test suite is further used as an alternative training and evaluation dataset in the REVEAL study [5] by Chakraborty *et al.*, who criticize the high number of duplicates compared to real-world datasets. We describe our approach to this matter in section 3.4.1.

## 3 Extracting an Assembly Language Dataset from the Juliet Test Suite

This section details the steps required to extract an assembly language dataset from the Juliet test suite that is suitable for processing by a machine learning model. The resulting dataset should fulfill the following requirements, which will serve as our design goals throughout the process:

**Coverage** The dataset should cover as large a fraction of the Juliet test suite as possible. While previous work [13, 22] focuses on subsets of the test suite to manage complexity, or uses static analysis rules to extract examples

[17], this work covers the largest possible fraction of the entire test suite that does not compromise the following two properties and is compatible with Linux environments.

**Context** While our goal is to provide examples on a function level, a single function can be vulnerable or not depending on the content of related functions or data. The dataset should provide appropriate context in terms of relevant functions for each example.

**No Label Leakage** Because of the very methodical construction of the test cases in Juliet, there are various ways in which label information can leak into examples. Symbols include label information in their names, *e.g.*, when they end in `good` or `bad`. Context can also leak information, *e.g.*, the presence of any context is enough to distinguish a primary good function from a primary bad function in certain flow variants. Label leakage can lead to overestimation of a machine learning model’s predictive performance during validation as the model exploits inadvertent correlations between examples and labels. Consequently, a model trained on a “leaky” dataset will exhibit worse performance in real-world applications, where these correlations do not exist. Hence, no label leakage should occur.

## 3.1 Preparations

Before we can extract examples, the Juliet test suite requires some preparations, which we discuss in the following.

### 3.1.1 Modifications

As a sanity check, the support file `io.c`, which we need to include for context, contains 18 empty functions by the names of `good1-good9` and `bad1-bad9`. We have to remove these functions from the support file because they would be present in every testcase after linking, and could be confused with actual examples. Ignoring empty functions entirely is not feasible, as there are testcases where empty functions are present on purpose, *e.g.*, concerning CWE-570 (Expression Always False). Hence, this modification is unavoidable. Aside from these support functions, we do not modify the Juliet source code in any way.

### 3.1.2 Compilation and Linking

We compile all testcases individually with GCC 11.2.0 using the provided Makefile generation script. Since our platform of choice is Linux, the script does not create Makefiles for Windows-specific CWEs such as CWE-247 (Reliance on DNS Lookups in a Security Decision). Compiling testcases individually results in single object files per translation unit, which we then link with the compiled support unit `io.c` because functions such as `printLine` are used in the testcases and could provide relevant context. Testcases consisting of multiple translation units are linked into one object file, together with the support unit.

Overall, this process results in object files for 41,812 testcases covering 91 CWEs, with the remainder consisting of Windows-specific testcases.

## 3.2 From C/C++ to Assembly Language

To obtain an assembly language representation of C/C++ sources, there are two obvious options. First, one can compile the sources directly into assembly language, skipping the assembly into machine code. Second, one can compile the sources fully into machine code, link them, and then disassemble the resulting binaries. We choose the latter approach, as it can in principle reveal more information about optimizations, *e.g.*, link-time optimization (LTO), as well as specific instruction encodings. It also more closely resembles a black-box analysis situation where source code is not available.

We disassemble each testcase’s linked object file using `objdump` configured to produce Intel syntax assembly. C++ symbols are demangled. Moreover, we extract static and dynamic symbol tables for each testcase to later distinguish global and local functions. The disassembly is parsed including addresses, binary representations, sections etc. and compared to the output of `capstone` on a per-instruction basis to ensure a correct disassembly.

## 3.3 Extracting Examples

In the following, we construct the ROMEO assembly language representation from the disassembly in a number of steps, including modifying operands and symbols, and selecting relevant context. Furthermore, we describe the mapping from testcases in Juliet to examples in the ROMEO dataset.

### 3.3.1 Symbol Representation

We build a scrambling table for each testcase, mapping each local symbol to a unique random name in the pattern of `1c000-1c999` to prevent label leakage, *e.g.*, from function names ending in `good` or `bad`. Global symbols are not renamed, as they contain no label information, but are crucial for more abstract vulnerability detection, *e.g.*, `memcpy`.

Many instructions contain addresses in the operands, which `objdump` annotates with corresponding symbols. As the specific memory layout is not important and potentially confusing for a classifier, we remove the addresses. Memory operands are represented only by symbols, which we replace with their scrambled names.

Moreover, many operands represent memory locations indirectly, *e.g.*, as offsets to registers. When `objdump` recognizes a known address, it emits a comment containing the address and a symbol plus offset. We replace the corresponding operand with the symbol and offset, also scrambled if applicable. The `lea` instruction in listing 1.2 illustrates this replacement.

### 3.3.2 Selecting and Representing Functions

Every function in the `.text` section of an object file is a potential example for ROMEO, or part of the context of one (see section 3.3.3). Hence, we extract a text representation of each function, consisting of a header line with the function’s (scrambled) name, followed by the disassembly modified in the aforementioned manner. The function name is prefixed with an exclamation mark, such that there is a unique token to mark the beginning of a function. Listing 1 shows an example of this representation.

However, not all functions are either positive or negative examples of a vulnerability. There are also supporting or completely unrelated functions in the object files. We only admit examples into the dataset that are primary or secondary good functions, or primary bad functions according to the regular expressions in [10]. The remained is ignored. We remove the primary good function to avoid label leakage (see the introduction to section 3). A large fraction, but not all, would be removed later in the process when duplicates are eliminated.

### 3.3.3 Including Context

The Juliet test suite lends itself to vulnerability classification on a function level, which we adapt for ROMEO. However, individual functions cannot always be classified based on their body alone. In terms of the source and sink model, a function containing a bad source might call another function containing the corresponding bad sink. Its vulnerability status can then only be determined by analyzing the called function as well.

To mitigate this issue, we include context information by concatenating the text representation of a given function with the text representation of all functions that are referenced by it, recursively. We exclude boilerplate and runtime functions such as `_libc_csu_fini` as their content is always identical. We further remove all bad functions if the given function is a primary or secondary good function, and vice versa, to avoid label leakage.

The scrambling of symbols applies to the context as well. Listing 1 shows an example of a function and the relevant context, where it can be seen that the argument passed to it is actually used in the context of an I/O operation.

## 3.4 Building a Dataset for Machine Learning

The collection of examples from the previous section is then labeled either with binary (“good” or “bad”) or multi-class (CWE number or “no weakness”) labels as chosen by the user. For our evaluation, we use the binary labels because most related work uses this formulation. Subsequently, we eliminate duplicates and split the dataset for proper validation.

### 3.4.1 Duplicate Elimination

Duplicates in benchmarks based on the Juliet test suite are observed by several works [3, 5, 20]. Since each testcase in Juliet is initially roughly unique, duplicates likely result from preprocessing or extraction steps, or in our case, from the compilation process. Our proposed ROMEO representation leads to a fraction of 2.6% (with context) and 9.7% (without context) duplicate examples, *i.e.*, examples with an identical representation.

For each set of identical examples, we remove all but one instance. Random selection determines which exact instance of a duplicate example is kept.

Our fraction of duplicates is lower compared to other works, *e.g.*, [20], which identifies an extreme value of 90.2% for Juliet. However, they do not remove the primary good functions of each testcase beforehand, which are mostly identical. Moreover, their representation is designed to require a very small vocabulary, and removes most natural language elements such as identifiers and literals. And most importantly, it does not consider context, which would help distinguish between similar instances. An overview of duplicate fractions in vulnerability detection benchmarks can be found in [5].

### 3.4.2 Splitting

We then randomly split the dataset into three parts for training, validation and testing, with a fraction of 80%, 10% and 10% of the examples, respectively. The resulting dataset is prepared for use by any machine learning methods capable of processing text sequences. We name it ROMEO to highlight its dependence on Juliet. It is publicly available at <https://gitlab.com/dlr-dw/romeo>.

## 3.5 Descriptive Statistics of ROMEO

This section is intended to give a brief overview of the distribution of samples in ROMEO. We distinguish between ROMEO and ROMEO without context, since the duplicate elimination process (see section 4) affects each variant differently. In table 1, we show the number of training, validation and test examples per weakness for the 20 most common weaknesses. There are 91 CWEs in total. However, the distribution of examples over CWEs is long-tailed, with CWE 190 accounting for approx. 11% of all examples. Flow variants are more uniformly distributed, hence the CWE distribution is more affected by the number of functional variants. Overall, the ROMEO variant including context consists of 134129 training examples, 16766 validation examples and 16765 held-out test examples. Without context, there are 124360, 15545 and 15544 examples, respectively. In the training set with context, there 34831 positive and 99298 negative examples. Without context, there are 34561 and 89799, respectively.

Table 1: Number of examples of top 20 weaknesses in ROMEO. Numbers differ depending on context inclusion because of subsequent duplicate elimination.

CWE	With Context			Without Context		
	Train	Val	Test	Train	Val	Test
190 Integer Overflow	14794	1947	1878	14784	1863	1910
122 Heap Based Buffer Overflow	11480	1418	1395	9959	1235	1215
191 Integer Underflow	11149	1355	1375	11097	1374	1360
762 Mismatched Mem. Mgmt.	10533	1301	1332	10485	1238	1323
121 Stack Based Buffer Overflow	8828	1091	1084	7885	978	976
590 Free Memory Not on Heap	6166	750	754	4932	604	617
401 Memory Leak	4990	622	630	4965	633	603
134 Uncontrolled Format String	4482	551	573	4460	577	550
457 Use of Uninitialized Variable	3773	450	502	3780	476	466
124 Buffer Underwrite	3471	465	406	3032	387	365
127 Buffer Underread	3441	469	436	3016	397	369
369 Divide by Zero	3299	399	419	3255	413	427
195 Signed-Unsigned Conv. Error	3211	384	399	2586	317	319
194 Unexpected Sign Extension	3164	410	420	2599	312	305
415 Double Free	2904	345	380	2929	319	369
400 Resource Exhaustion	2811	312	342	2766	347	345
126 Buffer Overread	2728	335	330	2471	290	287
36 Absolute Path Traversal	2668	341	330	2327	303	296
23 Relative Path Traversal	2651	326	361	2524	304	341
78 OS Command Injection	2647	332	313	2469	312	281
Total	134129	16766	16765	124360	15545	15544

```

!1c383:
push rbp
mov rbp, rsp
sub rsp, 0x10
mov DWORD PTR [rbp-0x4], 0x0
mov DWORD PTR [rbp-0x4], 0x0
mov eax, DWORD PTR [rbp-0x4]
sub eax, 0x1
mov DWORD PTR [rbp-0x8], eax
mov eax, DWORD PTR [rbp-0x8]
mov edi, eax
call 1c188
leave
ret

!1c188:
push rbp
mov rbp, rsp
sub rsp, 0x10
mov DWORD PTR [rbp-0x4], edi
mov eax, DWORD PTR [rbp-0x4]
mov esi, eax
lea rdi, _IO_stdin_used+0x6e
mov eax, 0x0
call printf
nop
leave
ret

```

(1.1) The extracted function.

(1.2) The context of the extracted function.

Listing 1: A function extracted from a Juliet testcase concerning CWE-191 (Integer Underflow) and its accompanying context. Both are in the text representation as described in section 3.3. This example illustrates one purpose of including context functions, namely to check whether a result is actually used, *e.g.*, in an API call.

## 4 Experiments

We consider the ROMEO dataset and representation itself our main contribution for this work. Still, there are research questions relating to the design goals of ROMEO and the usefulness of its representation for vulnerability detection applications that should be answered empirically. We identify the following research questions:

- RQ1** What benefits and drawbacks are associated with the inclusion of context (as described in section 3.3.3)?
- RQ2** To what extent, if any, does our assembly language representation leak label information?
- RQ3** How does an off-the-shelf model using our assembly language representation compare to other methods, including ones with access to the C/C++ source code?

In the remainder of this section, we first describe our experimental setup including datasets and baselines. We then present our results structured along the aforementioned research questions and offer brief discussions. Finally, we summarize the results, linking questions and answers, and address the limitations of this evaluation.

## 4.1 Setup

This section describes the situation concerning binary vulnerability detection datasets derived from Juliet as well as our Transformer-based vulnerability detector.

### 4.1.1 Datasets

In section 2, we list several works that perform vulnerability detection on assembly or machine code representations of the Juliet test suite. In particular, there is [11], which is only available in an already encoded vector form, from which the original instructions cannot be recreated. The same is true for [13]. BVATT [1, 2], is not available publicly anymore, against the claims in their work.

In all cases, the respective authors did not accommodate our request for the datasets in their original form.

### 4.1.2 Baselines

While we cannot directly compare our representation combined with the Transformer based approach (see section 4.1.3) to other methods on identical data (due to availability reasons explained in section 4.1.1), we can draw conclusions from comparisons to other work on reasonably similar data. Specifically, we compare our approach to works that evaluate their methods on test sets derived from the Juliet test suite, namely Instruction2Vec [13, 12], BVDetector [22], Russel *et al.* [20], and REVEAL [5]. We modify the ROMEO dataset to match the respective subset or construction as closely as possible in each case.

### 4.1.3 Transformer-based Model for Vulnerability Detection

The empirical part of this work focuses mainly on exploring the advantages and drawbacks of an assembly language representation for vulnerability detection. Hence, we select a rather generic, but very powerful approach to perform the actual classification task, namely Transformers [23]. Specifically, we use the pre-trained CodeBERT model [9] for initialization. Our implementation is based on PyTorch [19] and HuggingFace Transformers [24].

Because assembly language is less complex and variable than the languages in the initial training set of CodeBERT (which does not include assembly directly), the included tokenization and encoding is not optimal. Hence, we replace them with a byte-pair encoding [7] optimized on the ROMEO training set, which can represent common mnemonics such as `mov` by a single token. With context, the average example is 318.4 tokens long, where the model can handle at most 512 tokens. Without context, it is 157.2 tokens.

In our comparison with other methods, we use the name “ROMEO method” to refer to the combination of our ROMEO assembly language representation and the Transformer model.

Table 2: Accuracy of our Transformer-based method on the held-out test set of ROMEO. We list the five CWEs where including context has the strongest positive or negative effect on overall accuracy, respectively. The number of examples is provided to assess the overall effect size.

CWE / Context:		Accuracy (%)		Examples (#)	
		w	w/o	w	w/o
190	Integer Overflow	98.3	89.2	1878	1910
191	Integer Underflow	98.1	88.7	1374	1360
762	Mismatched Mem. Mgmt.	97.9	89.9	1332	1326
122	Heap Based Buffer Overflow	95.7	88.2	1395	1215
401	Memory Leak	96.8	89.5	631	604
676	Use of Potentially Dangerous Func.	88.9	95.2	6	5
468	Incorrect Pointer Scaling	75.0	82.4	8	17
396	Catch Generic Exception	73.0	77.2	18	19
590	Free Memory Not on Heap	99.7	99.8	753	619
398	Poor Code Quality	94.3	96.6	64	59
Overall/Total		96.9	90.2	16764	15544

We train the model for ten epochs on the ROMEO training set with and without context, respectively. One epoch equals one iteration over the entire training set. We use a minibatch size of 16, a learning rate of 1.1e-5, and a L2 regularization coefficient of 3e-4. These values are determined using the validation set, and subsequent evaluations are performed on the held-out test set. Each training and evaluation is performed three times, and all reported results are the average over all runs, with standard deviations displayed where applicable.

## 4.2 Results

In the following, we present our experimental results. Each section addresses one of the three research questions from the beginning of section 4, in order. All metrics are reported in terms of a binary classification problem (vulnerable / not vulnerable) for better comparison with other methods. However, the ROMEO dataset can easily be configured by users to include multi-class classification labels, as can the ROMEO method.

### 4.2.1 Context

RQ1 asks “what benefits and drawbacks are associated with the inclusion of context?” We answer this research question quantitatively and qualitatively using two variants of the ROMEO dataset, one with context and one without. Context is defined in section 3.3.3.

We first apply our ROMEO method (see section 4.1.3) to both variants of the ROMEO dataset. With context, the overall accuracy on the held-out test

Table 3: Accuracy of our Transformer-based method on the held-out test set of ROMEO. We list the five flow variants, where including context has the strongest positive or negative effect on overall accuracy, respectively. The number of examples is provided to assess the overall effect size.

Flow Variant / Context:	Acc. (%)		Ex. (#)	
	w	w/o	w	w/o
62 Data flows using a C++ reference from one function to another in different source files	99.2	66.2	237	203
42 Data returned from one function to another in the same source file	98.6	65.7	236	188
61 Data returned from one function to another in different source files	99.0	70.0	242	217
43 Data flows using a C++ reference from one function to another in the same source file	97.7	65.9	219	204
83 Data passed to a class constructor and destructor by declaring the class object on the stack	92.6	72.3	332	325
8 <code>if(staticReturnsTrue())</code> and <code>if(staticReturnsFalse())</code>	98.7	98.4	501	516
15 <code>switch(6)</code> and <code>switch(7)</code>	99.3	99.0	451	442
14 <code>if(globalFive==5)</code> and <code>if(globalFive!=5)</code>	99.3	99.2	511	504
5 <code>if(staticTrue)</code> and <code>if(staticFalse)</code>	99.1	99.1	501	493
21 Flow controlled by value of a static global variable. All functions contained in one file.	99.3	99.8	304	317
Overall/Total	96.9	90.2	16764	15544

set is 96.9% and the overall F1 score is 94.0%. Without context, the accuracy and F1 score are 90.2% and 81.9%, respectively. Hence, on average, our method strongly benefits from the included context.

From a machine learning point of view, removing the context information can be interpreted as manual feature selection, which is occasionally done on purpose to prevent overfitting. Removing features that are unrelated to the problem prevents the classifier from adapting to spurious correlations. However, in our case, there are clearly instances where context is crucial to determine whether a function is vulnerable or not, *e.g.*, when the actual vulnerability is spread over multiple function calls. Still, context information might not always be relevant and introduce misleading features. To get a clearer picture of this situation, we evaluate our method for each flow variant and CWE individually.

Table 3 shows the accuracy obtained with and without context information and which flow variants are most affected. In line with our expectations, the flow variants that benefit most from including context all describe vulnerabilities spread over multiple functions or even multiple translation units. Here, we see accuracy improvements up to 34.5 percent points. Without context, it is not possible to identify whether a function is calling a “bad sink” (assuming no label leakage through symbols or inlining by the compiler). We also show the flow variants that are most negatively affected, and in the worst case, the difference in accuracy is less than two percent points.

In table 2, we present the CWE types most affected by the inclusion of context. Overall, the most positively affected CWE types are simply those with the most examples and vice versa for the most negatively affected types. The most negatively affected weakness types have such low representation in Juliet that we cannot extract a meaningful interpretation of the results. In contrast to the flow variants, there is no set of CWE types that specifically benefit from context inclusion – most of them do.

To answer RQ1, including context in the assembly language representation of ROME0 is clearly beneficial for vulnerability detection. The benefits and drawbacks of including context do not appear to be specific to certain types of weaknesses in terms of CWEs. They are specific to certain presentations of vulnerabilities, *i.e.*, flow variants, that are spread out over multiple functions or translation units.

#### 4.2.2 Label Leakage

RQ2 asks “To what extent, if any, does our assembly language representation leak label information?” Preventing label leakage is also one of the design goals of ROME0 (see section 3).

Our process is carefully designed to prevent label leakage in terms of features, *e.g.* by renaming symbols (section 3.3.1) and excluding telltale functions from the context (section 3.3.3). Still, it is theoretically possible that some occurrence of label leakage is missing. However, we can estimate the label leakage in ROME0 empirically. The Juliet test suite includes examples of CWE-546 (suspicious comment), whose vulnerability status is impossible to infer in our

Table 4: Accuracy and F1 score on the held-out test set of ROMEO with and without context, compared to other methods on their respective variants of Juliet. *Note that Russel et al. works on slices, while ROMEO and REVEAL work on functions.*

Method	Dataset	Accuracy (%)	F1 (%)
ROMEO	ROMEO w/o context	90.2 ± 0.2	81.9 ± 0.4
ROMEO	ROMEO	96.9 ± 0.2	94.0 ± 0.4
Russell <i>et al.</i>	Juliet (slices)	—	84.0
REVEAL	Juliet (functions, no SMOTE)	—	93.7

Table 5: Accuracy and F1 score on subsets of the held-out test set of ROMEO, compared to BVDetector on similar subsets of the Juliet test suite. *Note that BVDetector works on slices, while ROMEO works on functions.*

Method	Dataset	Accuracy (%)	F1 (%)
ROMEO	ROMEO (MC)	<b>95.6 ± 0.5</b>	<b>91.3 ± 1.1</b>
BVDetector	Juliet (MC, slices)	94.8	85.4
ROMEO	ROMEO (NH)	<b>98.1 ± 0.1</b>	<b>96.1 ± 0.2</b>
BVDetector	Juliet (NH, slices)	97.6	92.2
ROMEO	ROMEO (MC+NH)	<b>97.1 ± 0.2</b>	<b>94.1 ± 0.5</b>
BVDetector	Juliet (MC+NH, slices)	96.7	89.9

assembly language representation, because the comments from C/C++ are not carried over. In the test set, there are six positive and 25 negative examples of this CWE. If there is no label leakage, the highest possible accuracy any method can achieve for this CWE is 80.6%, by constantly making negative predictions. Our method reaches 96.9% accuracy on the whole dataset, but achieves only 77.4% on CWE-546, which is below this threshold and a strong indication that there is no exploited label leakage.

However, label leakage can also be present on a dataset level even if the individual examples are free of it. While ROMEO does not have a time component, there are three distinct groupings that should be considered during the sampling process, namely CWEs, flow variants, and functional variants. For our evaluation, we require all CWEs and flow variants to be represented proportionally in all splits, which precludes splitting along them. Testing generalization across CWEs or flow variants could be an interesting investigation, but is beyond the scope of this work. Furthermore, we do not split the dataset along the functional variants because it would hinder our comparison with other work (see section 4.2.3), all of which apply random sampling.

To summarize: empirically, our representation does not leak label information.

### 4.2.3 Comparison with Other Methods

RQ3 asks “How does an off-the-shelf model using our assembly language representation compare to other methods, including ones with access to the C/C++ source code?”

As alluded to in section 4.1.2, we can only offer an estimate of relative performance. All performance numbers that we use from other works are based on the same Juliet test suite, but there are different evaluation processes. The works differ in granularity, *i.e.*, program slices [22] vs. functions [13, 20, 5]. Moreover, we extract sample functions from the full dataset unconditionally, but popular options include a selection of candidates by static analysis. Deduplication is performed at different stages or according to different representations [20, 5], or not at all [13, 22]. With these limitations in mind, we offer comparisons to related methods [13, 20, 22, 5] and modify our dataset to match their evaluations as closely as possible.

**Instruction2Vec + Text-CNN** This combination of methods [13, 12] is evaluated on a small subset of the Juliet test suite, namely vulnerabilities of type CWE-121 (stack-based buffer overflows). It also relies on assembly language instructions instead of source code. For comparison, we evaluate our Transformer-based method on this single vulnerability category only. Our ROMEO method achieves an accuracy of 97.3%, compared to 96.1% reported by Lee *et al.* Both methods operate on a function level.

**Russel *et al.*** [20] employs a bespoke C/C++ lexer to obtain a token sequence directly from the source code. On the Juliet suite, which is not subsampled in their work, they report an F1 score of 84.0% for the best combination of classifiers. In contrast, the ROMEO method obtains an F1 score of 94.0% over the whole suite (see also table 4).

**BVDetector** [22] extracts program slices instead of functions, but operates on assembly language instructions. The authors evaluate their method on a subset of Juliet involving memory corruption (MC) and number handling (NH) vulnerability types as defined by the STONESOUP program<sup>1</sup>. We evaluate our ROMEO method on a subset corresponding to these types and obtain an accuracy of 97.1% on MC and NH combined, where BVDetector reaches 96.7%. Both subsets are evaluated individually in table 5.

**ReVeal** [5] is proposed alongside an in-depth analysis of the current state of deep learning-based vulnerability detection and an eponymous dataset. It provides interesting insight into the limitations of synthetic datasets such as Juliet, but nevertheless the authors evaluate their method on it. On the unmodified dataset, they report an F1 score of 93.7%, compared to ROMEO’s very close 94.0% (see also table 4). However, due to the extreme class imbalance observed

---

<sup>1</sup><https://samate.nist.gov/SARD/around.php>

by them, they apply SMOTE to alter the distribution by resampling that data and increase their F1 score to 95.7%.

Overall, the ROMEO method, consisting of our assembly language representation combined with an off-the-shelf Transformer approach, performs better than other state-of-the-art methods, except for REVEAL-SMOTE, to the degree that these methods can be compared fairly. Importantly, the outperformed methods include [20, 5], which have access to the full C/C++ source code including comments.

### 4.3 Limitations of this Study

There are aspects in which this investigation is limited. For example, our evaluation is performed on the function level, similar to [13, 20, 5]. This design choice is appropriate for the Juliet test suite and our derived dataset ROMEO, but may not translate well to real-world applications. This limitation applies only to our dataset, since our representation could in principle be annotated on a per-instruction level. Alternatively, explainable methods such as IVDetect [15] could be applied to generate per-line results even from per-method annotations.

We maintain that the Juliet test suite is a sensible choice for our investigation as it allows for a methodical and detailed evaluation of our representation. It also covers a very wide range of weaknesses. However, it is not representative of real-world software projects in terms of vulnerability statistics and code complexity.

Moreover, in its present state, the examples in the dataset only include information from `.text` sections. Hence, information from data segments is missing. In listing 1, this can be seen in the context function `lc188`. The format string handed to `printf` is not part of the representation.

Currently, comments from the C/C++ code are also not carried over to the assembly language representation. While this could be solved by accessing debug information, it is not reasonable to expect this information to be present in a real-world binary analysis scenario. Our choice of x86-64 ISA is another minor limitation, however, our representation is not technically restricted to it and could be adapted, *e.g.*, to ARM.

We discuss proposed solutions to these limitations in section 5.1.

### 4.4 Summary of Results

In the following, we provide a brief summary of the research questions posed and answered in this evaluation:

**RQ1** What benefits and drawbacks are associated with the inclusion of context (as described in section 3.3.3)?

**RA1** Including context in the assembly language representation of ROMEO is clearly beneficial for vulnerability detection. The benefits are specific to certain presentations of vulnerabilities that are spread out over multiple functions.

**RQ2** To what extent, if any, does our assembly language representation leak label information?

**RA2** Empirically, our representation does not leak label information.

**RQ3** How does an off-the-shelf model using our assembly language representation compare to other methods, including ones with access to the C/C++ source code?

**RA3** It performs better than other state-of-the-art methods (including C/C++-based), except for REVEAL-SMOTE, to the degree that these methods can be compared fairly.

## 5 Conclusion

In this work, we present ROMEO, a publicly available, reproducible and reusable binary vulnerability detection benchmark dataset. It is derived from the Juliet test suite [10] of C/C++ vulnerabilities by compiling and extracting an assembly language representation.

Our representation incorporates context information in the form of related functions, which allows for detection of cross-function vulnerabilities. It also includes symbols to relate API call semantics. However, symbols that would leak label information are replaced with random names.

To evaluate the capabilities of this representation, we combine it with an off-the-shelf Transformer model to build the ROMEO method, and compare the combination with other methods. We show that even state-of-the-art methods that have full access to the C/C++ code only outperform ROMEO in specific cases. Furthermore, we find that, empirically, there is no label leakage in the ROMEO dataset or caused by our representation. Finally, there is a clear benefit to including context functions derived from the call graph for types of vulnerabilities that are span multiple functions.

In the following, we provide a brief outlook regarding possible future research.

### 5.1 Future Work

First and foremost, future work should address the limitations mentioned in section 4.3. Low-hanging fruit includes adding data to the context instead of only code. Currently, only the `.text` segment is analyzed. This analysis could be extended to `.data`.

Insight could be gained from adding source code comments to the appropriate location in the assembly representation and adding further details using debug information. However, this needs to be done carefully as to not introduce label leakage. A further interesting evaluation would involve repeating the experiments on a different ISA, *e.g.*, ARM, and observing the qualitative effects.

On a larger scale, our process of deriving a dataset from binaries could be adapted to real-world vulnerabilities, *e.g.* from the NVD. However, it is not trivial to construct a dataset of C/C++ code vulnerabilities that can be fully compiled, which is a requirement for binary vulnerability detection. Alternatively, one could adapt an existing dataset, *e.g.*, Big-Vul [8], and accept that only a subset is provided in a readily compilable form.

## Availability

We provide the entire dataset, including the raw binaries and source code to generate them and the representation, and to reproduce all results in this work. The dataset is available for download at <https://gitlab.com/dlr-dw/romeo>.

## References

- [1] Afanador, K.N., 2021. A Benchmark Framework and Support for At-scale Binary Vulnerability Analysis. Ph.D. thesis. Naval Postgraduate School. Monterey, CA.
- [2] Afanador, K.N., Irvine, C., 2020. Representativeness in the benchmark for vulnerability analysis tools (B-VAT), in: USENIX Workshop on Cyber Security Experimentation and Test (CSET 20).
- [3] Allamanis, M., 2019. The adverse effects of code duplication in machine learning models of code, in: SPLASH Onward! [arXiv:1812.06469](https://arxiv.org/abs/1812.06469).
- [4] Balakrishnan, G., Reps, T., 2010. WYSINWYX: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems* 32, 1–84. doi:[10.1145/1749608.1749612](https://doi.org/10.1145/1749608.1749612).
- [5] Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* doi:[10.1109/tse.2021.3087402](https://doi.org/10.1109/tse.2021.3087402).
- [6] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation, in: *Empirical Methods in Natural Language Processing (EMNLP)*. [arXiv:1406.1078](https://arxiv.org/abs/1406.1078).
- [7] Devlin, J., Chang, M.W., Lee, K., Toutanova, K., . BERT: Pre-training of deep bidirectional transformers for language understanding, in: *arXiv preprint arXiv:1810.04805*.
- [8] Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries, in: *Mining Software Repositories (MSR)*, ACM. doi:[10.1145/3379597.3387501](https://doi.org/10.1145/3379597.3387501).

- [9] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages, in: EMNLP 2020, ACL. doi:[10.18653/v1/2020.findings-emnlp.139](https://doi.org/10.18653/v1/2020.findings-emnlp.139).
- [10] Juliet, 2012. Juliet test suite v1.2 for c/c++ user guide.
- [11] Le, T., Nguyen, T., Le, T., Phung, D., Montague, P., Vel, O.D., Qu, L., 2019. Maximal divergence sequential autoencoder for binary software vulnerability detection, in: International Conference on Learning Representations (ICLR). URL: <https://openreview.net/forum?id=ByloIiCqYQ>.
- [12] Lee, Y., Kwon, H., Choi, S.H., Lim, S.H., Baek, S.H., Park, K.W., 2019. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN. Applied Sciences 9, 4086. doi:[10.3390/app9194086](https://doi.org/10.3390/app9194086).
- [13] Lee, Y.J., Choi, S.H., Kim, C., Lim, S.H., Park, K.W., 2017. Learning binary code with deep learning to detect software weakness, in: International Conference on Internet (ICONI).
- [14] Li, X., Feng, B., Li, G., Li, T., He, M., 2021a. A vulnerability detection system based on fusion of assembly code and source code. Multimodality Data Analysis in Information Security 2021, 1–11. doi:[10.1155/2021/9997641](https://doi.org/10.1155/2021/9997641).
- [15] Li, Y., Wang, S., Nguyen, T.N., 2021b. Vulnerability detection with fine-grained interpretations, in: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE), ACM. doi:[10.1145/3468264.3468597](https://doi.org/10.1145/3468264.3468597).
- [16] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021c. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing doi:[10.1109/tdsc.2021.3051525](https://doi.org/10.1109/tdsc.2021.3051525).
- [17] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection, in: Network and Distributed System Security Symposium (NDSS), Internet Society. doi:[10.14722/ndss.2018.23158](https://doi.org/10.14722/ndss.2018.23158).
- [18] Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space, in: International Conference on Learning Representations (ICLR).
- [19] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library, in: arXiv preprint arXiv:1912.01703.

- [20] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning, in: International Conference on Machine Learning and Applications (ICMLA), IEEE. doi:[10.1109/icmla.2018.00120](https://doi.org/10.1109/icmla.2018.00120).
- [21] The MITRE Corporation, 2021. Common Weakness Enumeration (CWE). URL: <https://cwe.mitre.org/>.
- [22] Tian, J., Xing, W., Li, Z., 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. Information and Software Technology 123. doi:[10.1016/j.infsof.2020.106289](https://doi.org/10.1016/j.infsof.2020.106289).
- [23] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need, in: Neural Information Processing Systems (NeurIPS). [arXiv:1706.03762](https://arxiv.org/abs/1706.03762).
- [24] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M., 2020. Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations, Association for Computational Linguistics.