Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

9-2010

An automated approach for finding variable-constant pairing bugs

Julia LAWALL
University of Copenhagen

David LO
Singapore Management University, davidlo@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

An Automated Approach for Finding Variable-Constant Pairing Bugs

Julia Lawall
DIKU, University of Copenhagen
INRIA-Regal
julia@diku.dk

David Lo
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

ABSTRACT

Named constants are used heavily in operating systems code, both as internal flags and in interactions with devices. Decision making within an operating system thus critically depends on the correct usage of these values. Nevertheless, compilers for the languages typically used in implementing operating systems provide little support for checking the usage of named constants. This affects correctness, when a constant is used in a context where its value is meaningless, and software maintenance, when a constant has the right value for its usage context but the wrong name.

We propose a hybrid program-analysis and data-mining based approach to identify the uses of named constants and to identify anomalies in these uses. We have applied our approach to a recent version of the Linux kernel and have found a number of bugs affecting both correctness and software maintenance. Many of these bugs have been validated by the Linux developers.

Categories and Subject Descriptors: D.2.4 [Software Engineering]:Software/Program Verification–Statistical methods

General Terms: Algorithms, Design, Experimentation
Keywords: Variable-Constant Pairing Bugs, Anomaly Detection, Clustering, Linux

1. INTRODUCTION

Integer constants are heavily used in operating systems code, in interpreting values read from devices, in constructing values to be written to devices, and in representing flags. Some constants are represented explicitly, as so-called "magic numbers". These are well-known to be extremely error-prone, because their form is essentially meaningless: in writing the code it is easy to mistype some digit, and in reading the code it is impossible to tell what concept is intended to be represented. Operating systems code thus often defines *named* constants, either using `#define` or an enumeration type declaration. In this way, a constant is associated with a name that suggests its value, and the programmer can

1. `mboxes = mcp->out_mb; ...  if (mboxes & BIT_0) ...` `mboxes = mcp->in_mb; ...  if (mboxes & BIT_0)`

2. `(endpoint->bEndpointAddress & USB_TYPE_MASK) ==` `USB_DIR_OUT`

3. `tp->tg3_flags2 & TG3_FLAG_10_100_ONLY`

Figure 1: Bugs in constant usage, in Linux 2.6.30, in drivers/scsi/qla2xxx/qla_mbx.c, drivers/net/wireless/zd1211rw/zd_usb.c, and drivers/net/tg3.c, respectively

use the constant without being aware of this value. Spelling mistakes are likely to result in a reference to a non-existent identifier, and the bug will be caught by the compiler.

Nevertheless, the use of `#define` and enumeration type constants is not sufficient to prevent all errors. Indeed, compilers provide little or no assistance in ensuring that a given constant is used in the right context. `#define` constants are seen by the compiler as the integer value that they expand into, and in the case of C, values declared in an enumeration type have type `int` [8]. Figure 1 illustrates the three kinds of errors that can occur. In the first example, the constant `BIT_0` has the right value, but the wrong name (*name bug*). The `out_mb` and `in_mb` fields should be combined with `MBX_0`, which has the same value, as done elsewhere in the file. In the second example, the value of `USB_TYPE_MASK` has the wrong value for the context in which it occurs (*value bug*). The result of the bit-and operation is compared to `USB_DIR_OUT`, which only has 1 bits in positions where `USB_TYPE_MASK` has 0 bits, and thus the result of the comparison is always false. Finally, in the third example, the right constant is used in the wrong context (*context bug*). The `TG3_FLAG_10_100_ONLY` constant should be used with the `tg3_flags` field, rather than the `tg3_flags2` field. Of these, a name bug does not affect the behavior of the program, but can harm its readability and future maintenance. A value or context bug produces incorrect behavior: for example, a condition reported by a device may never be detected, or a device may be provided with an inappropriate value. Some of these bugs may be detected quickly, *e.g.*, if a device exhibits unexpected behavior. Others may linger, due to the difficulty of finding such bugs by manual inspection and the lack of appropriate tools.

Essentially, the problem of ensuring that the right constant is used in the right context is a type checking problem. Indeed, a group of constants that may be used in a particular kind of context may be considered to form a type. Nevertheless, the problem of inferring these types is more difficult than

the problem of ordinary type inference, because there are no kinds of terms on whose types we can rely. Thus, we must first identify what the types are, and then check that they are respected by the code that uses these values. This problem is further complicated by issues of dependent types, where the type expected in one function argument or structure field depends on the value contained in another one, of subtypes, where one module may define supplementary constants that are intended to be used in a context that expects constants of another existing type, and of bugs, implying that any given use of a constant cannot be assumed to be a correct use, and thus cannot in itself imply a type definition.

In this paper, we propose to address the issue of type inference for named constants using a combination of program analysis and data mining. In our approach, program analysis is used to collect information about the use of constants, and data mining is used to filter out probable anomalous uses to be able to classify the constants into types. We have applied our approach to the Linux 2.6.30 kernel,[1] focusing on constants that are used in bit-and and bit-or operations, representing the processes of extracting information from existing values and of constructing new values. In Linux 2.6.30, we have found 10 probable bugs, of which 7 have been confirmed by Linux developers.[2] This number of bugs is not large, but the ability to find these bugs is unique to our approach; existing bug-finding approaches typically focus on sets or sequences of function calls [5, 11, 21, 24, 26], and thus are not able to find bugs in the use of constants.

Concretely, the contributions of this paper are as follows:

- We identify the pattern of named constant usage in Linux code, providing a basis for bug finding.

- We define a program analysis that provides insight into the flow of named constants through Linux code.

- We define a data mining strategy based on clustering that groups named constants into types. This approach to clustering is novel in that it has no *a priori* knowledge of the number of clusters, but is efficient enough to be used with large and varied data sets.

- We provide a detailed evaluation of our approach on the complete Linux 2.6.30 source code.

- We classify some forms of named constant usage that are problematic for our approach, thus providing a basis for future work.

The rest of this paper is organized as follows. Section 2 presents the program analysis used to collect information about constant usage. Section 3 presents some aspects of the implementation of this analysis that are needed to address specific issues occurring in Linux code. Section 4 presents the data mining techniques used first to group the constants into clusters, representing types, and then to identify probable bugs in constant usage. Then, Section 5 evaluates our results on Linux 2.6.30, Section 6 presents related work, and Section 7 concludes.

---

[1]This is a recent version, but one that was released before we had submitted any patches based on our work.

[2]The reactivity of the Linux maintainers varies, and thus we have sometimes received no response to our enquiry.

2. ANALYSIS OVERVIEW

Constants are used within the Linux kernel to represent internal flags, indicating various conditions, and to interact with external devices, which communicate using bit sequences of various sizes. For efficiency, bits having different purposes are often packed into a single unit of addressable memory (byte, word, etc.). Accessing information from such bit sequences is carried out using the bit-and operator &, as previously illustrated in Figure 1, and bit sequences are constructed using the bit-or operator |. In this paper, we focus on these operations. Constants are also involved in equality tests and shift operations. We leave the extension of our approach to these operators as future work.

Constants are typically first used to extract components using the bit-and operator from some values that we designate as *sources*, then are transmitted through the code structure by the use of various assignments, and are finally used to construct new values with the bit-or operator to be passed to locations we designate as *sinks*. The goal of our analysis is to associate constants to the sources and sinks with which they interact. Sources and sinks can in principle be any sort of expression, but to simplify the presentation, we assume that they are specially designated variables. In the next section, we will instantiate sources and sinks as structure fields. The analysis is intraprocedural and flow-insensitive. It does not take aliases into account. In our implementation, we provide flow-sensitivity for local variables via prior conversion to Static Single Assignment (SSA) form [1] and a weak form of alias analysis via types, as described in the next section.

2.1 Syntax

We present the analysis in terms of the simple imperative language defined in Figure 2. The actual implementation, however, handles full C code, as described in the next section. In the language of Figure 2, a program consists of an unordered set of assignments of variables to expressions, where the lack of ordering reflects the flow insensitivity of the analysis. An expression is either a variable, a constant, a bit-and operation or a bit-or operation. Some variables are designated as *sources* or *sinks*.

$$c \in \mathsf{Constants} \qquad source \in \mathsf{Sources} \subseteq \mathsf{Variables}$$
$$v \in \mathsf{Variables} \qquad sink \in \mathsf{Sinks} \subseteq \mathsf{Variables}$$

$$prog \in \mathsf{Programs} \quad ::= \mathcal{P}(\mathsf{Statements})$$
$$stmt \in \mathsf{Statements} ::= v\ \mathtt{=}\ expr$$
$$expr \in \mathsf{Expressions} ::= c \mid v \mid expr\ \mathtt{\&}\ expr \mid expr\ \mathtt{|}\ expr$$

Figure 2: Syntax

2.2 Analysis

The analysis collects an *environment* containing information about the bindings of variables, to propagate this information between the various statements, and at the same time uses the information in the environment to generate an *output* describing the interaction between constants and either sources or sinks. For example, if we consider the case where sources and sinks are structure fields, then for the first line of case 1 of Figure 1, the environment would contain a mapping of `mboxes` to `mcp->out_mb` and the output would indicate that `mcp->out_mb` interacts with `BIT_0`. The analysis iteratively accumulates the environment and output until reaching a fixed point.

The semantic domains used by the analysis are shown in Figure 3. These are *representations*, *summaries*, *environments*, and *output*. A representation r is a pair of a set of variables and a set of constants. These are the variables and constants that contribute in a particular way to the value of an expression. A summary s provides the complete information collected about the computation performed by an expression as a tuple of three representations: 1) one indicating the set of variables and constants to whose value the expression may evaluate, 2) another indicating the variables and constants involved in any bit-and operation that is used to compute the value of the expression, and 3) a third providing the same information for bit-or operations. For example, in case 1 of Figure 1, the summary corresponding to `mboxes & BIT_0` would be $\langle\langle\emptyset,\emptyset\rangle,\langle\{\texttt{mboxes},\texttt{mcp->out\_mb}\},\{\texttt{BIT\_0}\}\rangle,\langle\emptyset,\emptyset\rangle\rangle$. An environment ρ maps variables to summaries. A binding in this environment is written as (v,s) and records the effect of assignment statements. Finally, an output σ is a pair of *relations* from sources and sinks, respectively, to constants. The constants in an output are annotated with the position (offset from the start of the program) of the interaction between the source or sink and the constant, and thus an output provides information about both the kind and number of such interactions.

$r \in representations = \mathcal{P}(\mathsf{Variables}) \times \mathcal{P}(\mathsf{Constants})$
$s \in summaries =$
$\qquad representations \times representations \times representations$
$\rho \in environments = \mathsf{Variables} \to values$
$\sigma \in output = (\mathcal{P}(\mathsf{Sources} \times (\mathsf{Positions} \times \mathsf{Constants})) \times$
$\qquad\qquad (\mathcal{P}(\mathsf{Sinks} \times (\mathsf{Positions} \times \mathsf{Constants}))$

Figure 3: Semantic domains used by the analysis

Representations are ordered as follows, where V_i is a set of variables and C_i is a set of constants:

$$\langle V_1, C_1 \rangle \sqsubseteq \langle V_2, C_2 \rangle \iff V_1 \subseteq V_2 \wedge C_1 \subseteq C_2$$

$\bot$ abbreviates the representation $\langle\emptyset,\emptyset\rangle$. Summaries are ordered such that a pair of summaries is related if all of their components are related. Environments are ordered similarly. In each case, the least upper bound operation $\sqcup$ is defined by computing the union of corresponding sets of variables and constants. Finally, output is ordered by the subset relation. Each of these orderings forms a complete lattice.

The rules for expressions are defined in Figure 4. These rules infer judgements of the form $\rho \vdash expr : s, \sigma$, where s is a summary containing information about the variables and constants that are used to compute the value of the expression and σ represents the output. In each rule, only information about the outermost kind of operator, bit-and or bit-or, is collected. This strategy is based on the observation that *e.g.* in `(a & B) | C` we do not know what named constant, if any, the parenthesized subexpression represents. It is for this reason that summaries contain separate components for `&` and `|` information.

Among the rules for expressions, only the rule for bit-and expressions generates output. Our understanding of bit-and is that it is used to extract information from sources. In this case, if a source is used in computing one of the bit-and arguments and a constant is used in computing the other, then the expression represents an interaction between them. The construction of the output in this case uses the function $\mathsf{source}(r)$ which returns the variables in the representation r

$$\rho \vdash v : \rho(v) \sqcup \langle\langle\{v\},\emptyset\rangle,\bot,\bot\rangle,\emptyset \qquad \rho \vdash c : \langle\langle\emptyset,\{c\}\rangle,\bot,\bot\rangle,\emptyset$$

$$\frac{\rho \vdash expr_1 : \langle r_{=_1}, r_{\&_1}, r_{|_1}\rangle, \sigma_1 \qquad \rho \vdash expr_2 : \langle r_{=_2}, r_{\&_2}, r_{|_2}\rangle, \sigma_2}{\begin{array}{l}\rho \vdash expr_1 \ \& \ expr_2 : \langle \bot, r_{=_1} \sqcup r_{\&_1} \sqcup r_{=_2} \sqcup r_{\&_2}, \bot\rangle,\\ \quad \{v \mapsto c \mid v \in \mathsf{source}(r_{=_1} \sqcup r_{\&_1}) \wedge c \in \mathsf{cst}(r_{=_2} \sqcup r_{\&_2})\} \cup \\ \quad \{v \mapsto c \mid v \in \mathsf{source}(r_{=_2} \sqcup r_{\&_2}) \wedge c \in \mathsf{cst}(r_{=_1} \sqcup r_{\&_1})\} \cup \sigma_1 \cup \sigma_2\end{array}}$$

$$\frac{\rho \vdash expr_1 : \langle r_{=_1}, r_{|_1}, r_{|_1}\rangle, \sigma_1 \qquad \rho \vdash expr_2 : \langle r_{=_2}, r_{|_2}, r_{|_2}\rangle, \sigma_2}{\rho \vdash expr_1 \ | \ expr_2 : \langle \bot, \bot, r_{=_1} \sqcup r_{|_1} \sqcup r_{=_2} \sqcup r_{|_2}\rangle, \sigma_1 \cup \sigma_2}$$

Figure 4: Analysis rules for expressions

that are sources, and the function $\mathsf{cst}(r)$ which returns the constants in the representation r. No output is generated for a variable or constant expression, because these do not involve any interactions. No output is generated for a bit-or expression because it expresses only the construction of a value, but not the communication of the constructed value to a sink. Our implementation also collects information about the use of `==` and `!=`. These operators add information to the first component of a summary, analogous to the rules for variables and constants.

The rules for statements are defined in Figure 5. These rules infer judgements of the form $\rho \vdash stmt : \rho', \sigma$, where ρ' contains information about the assigned variable and σ represents the output. If the left-hand side variable is not a sink, the only effect is to extend the environment with a binding of the variable to the value obtained by analyzing the expression. If the variable is a sink, then output is also generated. This output maps the sink to each possible constant in the bit-or information contained in the summary resulting from analyzing the right-hand side expression *expr*.

$$\frac{v \notin \mathsf{Sinks} \qquad \rho \vdash expr : s, \sigma}{\rho \vdash \texttt{v = } expr : \rho \sqcup \{(\texttt{v}, s)\}, \sigma}$$

$$\frac{v \in \mathsf{Sinks} \qquad \rho \vdash expr : \langle r_=, r_\&, r_|\rangle, \sigma}{\rho \vdash \texttt{v = } expr : \rho \sqcup \{(\texttt{v}, \langle r_=, r_\&, r_|\rangle)\}, \{\texttt{v} \mapsto c \mid c \in \mathsf{cst}(r_|)\} \cup \sigma}$$

Figure 5: Analysis rules for statements

The rule for programs iterates the rules for statements over the set of statements in the program until the resulting environment and output reach a fixed point. This iteration terminates because environments and outputs form a complete lattice, and because each iteration only monotonically adds information to each of these entities. The final result is the output at the end of this iteration.

2.3 Example

To illustrate the analysis, we consider the program shown below, where `source` is a source, `sink` is a sink, `x`, `y`, and `z` are neither sources nor sinks, and `A`, `B`, `C`, `D` are constants.

```
1.  x = source & A      3.  z = y | C
2.  y = x & B           4.  sink = z | D
```

To emphasize the flow-insensitive nature of the analysis, we first analyze all of the expressions based on the initial environment, then analyze the enclosing statements, and then iterate. The analysis steps are shown in Table 1. Each row in the table corresponds to one of the above statements. Within each row, the top line is the resulting value or environment, as appropriate, and the bottom line is the added output (positions are elided). A column labelled "exp" contains the

	exp	stmt	exp	stmt
1. `x = source & A`	$a_{1,1} = \langle \bot, \langle \{\mathtt{source}\}, \{\mathtt{A}\}\rangle, \bot\rangle$ `source ↦ A`	$\{(\mathtt{x}, a_{1,1})\}$ none	$a_{1,2} = a_{1,1}$ none	$\{(\mathtt{x}, a_{1,2})\}$ none
2. `y = x & B`	$a_{2,1} = \langle \bot, \langle \{\mathtt{x}\}, \{\mathtt{B}\}\rangle, \bot\rangle$ none	$\{(\mathtt{y}, a_{2,1})\}$ none	$a_{2,2} = \langle \bot, \langle \{\mathtt{source}, \mathtt{x}\}, \{\mathtt{A}, \mathtt{B}\}\rangle, \bot\rangle$ `source ↦ B`	$\{(\mathtt{y}, a_{2,2})\}$ none
3. `z = y | C`	$a_{3,1} = \langle \bot, \bot, \langle \{\mathtt{y}\}, \{\mathtt{C}\}\rangle\rangle$ none	$\{(\mathtt{z}, a_{3,1})\}$ none	$a_{3,2} = a_{3,1}$ none	$\{(\mathtt{z}, a_{3,2})\}$ none
4. `sink = z | D`	$a_{4,1} = \langle \bot, \bot, \langle \{\mathtt{z}\}, \{\mathtt{D}\}\rangle\rangle$ none	$\{(\mathtt{sink}, a_{4,1})\}$ `sink ↦ D`	$a_{4,2} = \langle \bot, \bot, \langle \{\mathtt{y}, \mathtt{z}\}, \{\mathtt{C}, \mathtt{D}\}\rangle\rangle$ none	$\{(\mathtt{sink}, a_{4,2})\}$ `sink ↦ C`

Table 1: Analysis trace

result of processing the right hand side of an assignment according to the environment that is the least upper bound of the environments generated by the previous column, if any. A column labelled "stmt" contains the result of processing the complete assignment according to the same environment.

The analysis reaches a fixed point after the two iterations shown in Table 1. The result is then the accumulated output: `source ↦ A` resulting from `source & A`, `source ↦ B` resulting from `x & B`, and `sink ↦ C` and `sink ↦ D` resulting from `sink = z | D`.

3. IMPLEMENTATION OF THE ANALYSIS

To successfully treat Linux code, the analysis must parse the source code, identify named constants, and select a notion of source and sink. In practice, we have also found it necessary to implement flow sensitivity for local variables.

Parsing. Tools that process C code typically first apply the C preprocessor to eliminate all preprocessor directives. This is, however, not appropriate in our case, as it eliminates the names of constants that are defined using `#define`. Furthermore, to collect a maximum of information about constant usage, we would like the analysis to consider as much of the source code as possible, including portions of code that are specific to the more obscure hardware configurations. To address these issues, we use the C parser of the Coccinelle program matching and transformation tool [18, 19], which parses C code without expanding macro definitions. This parser can parse around 97% of the Linux 2.6.30 kernel.

Identification of named constants. Linux constants typically have names that are constructed entirely of capital letters. Nevertheless, this strategy is not always followed. Furthermore, some constants may be defined in multiple files, potentially with a different meaning in each case. To be able to identify constants accurately, the analysis initially collects for each file a list of the constants that they define, and the position in the file of that definition. Subsequently, in processing each `.c` file, the analysis phase recursively unwinds all of the `include` directives, collecting for each included file the set of named constants that it defines. Finally, the constants defined by the `.c` file are added to this set. This process does not take into account `#ifdef` directives, and thus information about all defined constants is available. A constant may be defined multiple times, under different `#ifdef`s, potentially leading to ambiguity. We assume that the multiple definitions may change the value of the constant, but not its purpose; this assumption has not lead to any problems in practice.

Selection of sources and sinks. We have chosen to use structure fields as our notion of both sources and sinks. Linux structures are heavily used to communicate complex information between different parts of the kernel, and thus their fields tend to have a fixed semantics. Indeed, we have found that a given field of a given structure type is often always used in the same way, regardless of the structure instance with which it is associated.[3] Thus, we choose to represent a structure field as a pair of the type of the structure and the name of the field, thus unifying the information collected for all occurrences of a given structure field. This indeed provides a weak form of alias analysis, as long as structures are used in a well-typed way.

This choice of the representation of structure fields raises the need to determine the type of each referenced structure. When the structure is referenced as a variable, its type can be obtained from the variable declaration, without knowing the structure definition. When the structure is expressed as a more complex expression, typically another structure field reference, the definition of the type of the containing structure is required to determine the type of the structure itself. For this, the pass that identifies constants also collects typedefs and structure declarations. This information is then used to infer the types of structure fields.

Implementation of flow sensitivity. Flow insensitive analysis is less expensive than flow sensitive analysis, and is typically sufficient when the tracked locations are mostly used in a uniform way. We have argued that this is often the case for structure fields. Nevertheless, we have found that it may not be the case for local variables in Linux code. Indeed, it is common to declare an integer-typed local variable with a generic name such as `data` and use it in for multiple purposes within a single, often large and complex, function.

Figure 11 illustrates a typical case.[4] The code is essentially divided into two regions, with the first extending from line 1 to line 9, and the second extending from line 11 to line 12. In the first part, the variable `flag` interacts with constants of the form `TDES1_*`, while in the second part, the variable `flag` interacts instead with constants of the form `TDES0_*`. In each case, the value of `flag` is ultimately stored in the same structure, but in fields having different purposes. In our data-mining based approach, it important to keep the various uses of the variable separated from each other. Otherwise, the two sets of constants could be merged, which would lead

---

[3] We will, however, revisit this assumption in Section 5.4.
[4] In this code, the structure field initializations involve the macro `cpu_to_le32`. This function, and others like it, affect only the bit order, and are considered by our implementation to be the identity function.

to overlooking a bug in the case of an interaction between them. And if the sets of constants are not merged, but the variables remain identical, then one of the initializations of a field of the `priv` structure (line 9 or line 12) would be reported as a bug, amounting to a false positive.

```
if (priv−>cur_tx − priv−>dirty_tx == priv−>tx_ring_size / 2)    1
  flag = TDES1_CONTROL_IC | TDES1_CONTROL_LS |                  2
       TDES1_CONTROL_FS;                                        3
else                                                            4
  flag = TDES1_CONTROL_LS | TDES1_CONTROL_FS;                   5
...                                                             6
if (entry == priv−>tx_ring_size − 1)                            7
  flag |= TDES1_CONTROL_TER;                                    8
priv−>tx_ring[entry].length = cpu_to_le32(flag | skb−>len);     9
                                                                10
flag = TDES0_CONTROL_OWN | (plcp_signal << 20) | 8;            11
priv−>tx_ring[entry].status = cpu_to_le32(flag);              12
```

drivers/net/wireless/adm8211.c::1630-1651

Figure 6: Illustration of the need for flow sensitivity in different regions of code

To address this problem, we adopt a standard approach to providing flow-sensitivity: conversion of the source program to Static Single Assignment (SSA) form [1]. In this form, local variables are renamed such that every local variable is defined at only one position in the source code. So-called ϕ functions are inserted at merge points, such as the end of a conditional, to collect the variables that can contribute to the value of each variable that is live after the merge point. This has the effect of renaming local variables, but has no impact on the set of structure fields and thus no impact on the set of sources and sinks considered by the analysis. In the result of the conversion to SSA form, the `TDES1_*` and `TDES0_*` constants are accumulated in different variables, and are transmitted separately to the different fields of the `priv` structure by the analysis.

4. MINING ALGORITHM

The goal of our mining algorithm is to discover anomalous variable-constant pairings automatically. It works in three steps as shown in Figure 7. The first step constructs a graph to represent the relationships between variables and the constants they are paired with by the analysis. The second step then uses this graph to cluster related constants together. Finally, based on these clusters, the third step detects anomalous variable-constant pairings.
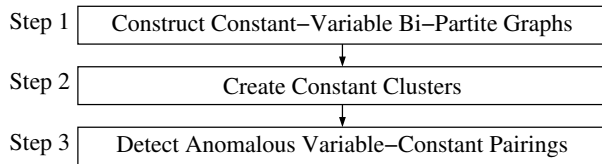
Step 1	Construct Constant−Variable Bi−Partite Graphs
Step 2	Create Constant Clusters
Step 3	Detect Anomalous Variable−Constant Pairings

Figure 7: High-level steps

Step 1. Graph Construction. Our goal is to construct a bi-partite graph capturing the relationships between constants and variables. One side of the graph contains all the constants, and the other side of the graph contains all the variables.[5] One side of the graph contains all the constants,

---
[5]In this section, we use "variable" to refer to sources and sinks, *i.e.*, structure fields in our case.
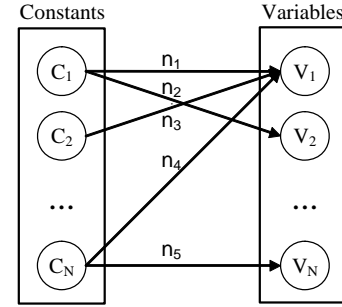

Figure 8: Constant-variable bi-partite graph

and the other side of the graph contains all the variables. There is an edge from a constant to a variable if the variable is used together with the constant. The label on the edge shows the number of times a constant is used with the variable. In the graph, we merge any variables V_1 and V_2, if the set of constants C_1 associated with V_1 is a subset of the set of constants C_2 associated with V_2. An example graph is shown in Figure 8.

Step 2. Create Constant Clusters. This step groups constants based on their *behavior usage profile i.e.*, the number of times they are paired with various variables. Our goal is to create clusters where all the constants in the cluster are used with a similar set of variables. Each cluster of constants can be viewed as a weak "type" *i.e.*, they are used in the same way.

Distance Metrics. To create a constant cluster, we first define a measure of distance between two constants. Each constant is associated with a corresponding behavior usage profile which is a vector containing an entry for each variable. An entry in this vector contains the number of times the constant is paired with the corresponding variable multiplied by a weight denoting the importance of that particular variable. This weight is determined by considering how many constants that variable is paired with in the dataset:

$$weight(v) = \frac{1}{|Constants\ paired\ with\ v|}$$

This strategy is based on the concept of inverse document frequency (or IDF) commonly used in information retrieval [14].

Based on this vector, metrics from information retrieval can be used. Of these metrics, we choose cosine similarity [14]. Cosine similarity performs normalization with respect to the size of the vector and is thus less sensitive to this size. Since the vectors tend to be sparse (i.e., a constant is normally used with only a few variables), cosine similarity is more accurate than other similarity metrics, including Euclidean and Jaccard similarity [6, 14]. Cosine similarity is defined as follows:

$$cos(V_1, V_2) = \frac{V_1 \bullet V_2}{|V_1||V_2|}$$

The numerator is the *dot product* of the two vectors, and the denominator is the product of the *magnitude* of the first vector, i.e., the square root of the dot product of the vector with itself, with that of the second vector. We then define distance as follows:

$$dist(V_1, V_2) = 1 - cos(V_1, V_2)$$

Consider for example two constants C_1 and C_2. C_1 is used with variables V_1 and V_2 while C_2 is used with variable V_2. The vectors corresponding to C_1 and C_2 are [1 1] and [0 1] respectively. The cosine similarity of the two vectors are $1/(\sqrt{2} \times \sqrt{1}) = 0.71$. The distance is equal to $1 - 0.71 = 0.29$.

As a cluster is a collection of constants, we can aggregate the constants' behavioral usage profile to form the profile of a cluster. The profile of a cluster is a vector where each entry is the sum of the values of the corresponding vector entries of all constants belonging to that cluster. The distance between a cluster and a constant, or a cluster and another cluster, is then computed in a similar way as the distance between a constant and another constant.

Clustering. Many existing clustering algorithms require specifying the number of clusters, which is not known in our case. We propose a new heuristic-based clustering algorithm that works without the need to know the target number of clusters in advance:

1. Create an initial set of clusters. This step greedily scans each constant and assigns it to the best cluster created so far whose distance is close enough based on a user-defined threshold θ. If no cluster is found, a new cluster is created.

2. Iteratively refine the set of clusters. After a temporary cluster is formed, we try to re-locate each data point to the cluster whose distance is closest. At each iteration, constants are moved to their respective nearest cluster. After this reshuffling, the profile of each cluster is recomputed. We repeat this process until a fixed point is reached and no constants can move to another cluster. In our experiments, a fixed point is reached in 3 iterations.

3. Merge clusters that are very close based on the user defined threshold θ.

The algorithm is described in more detail in Figure 9. We use 0.35 as the value of θ. We determined the thresholds empirically based on a number of good clusters that we knew in advance. The threshold of 0.35 keeps many of these good clusters. We compare our algorithm to some existing clustering algorithms in Section 6.

Step 3. Anomaly Detection. In the third step, we detect anomalies by looking for variables that are shared by more than one cluster. On finding an anomaly, a bug report, consisting of a variable V, a constant C, and a cluster L, denoted as $\langle V, C, L \rangle$, is generated. We rank our bug reports based on their suspiciousness. It is our intuition that suspiciousness of a variable V being paired to a constant C in a cluster L is related to the following:

Size of the cluster L. The size of a cluster is the number of constants that it contains. We denote the size as $size(L)$. It is our intuition that the more elements a cluster L has, the more likely a programmer is to make a mistake in using the constant in L.

Strength of the association between the variable V and the cluster L containing C. Let us define $freq(V, C)$ to be the number of times a variable V is paired to a constant C. We define the strength of this association as follows:

```
Procedure ClusterConstants
Inputs:
    CONSTANTS : Set of constants to be clustered
    θ₁ : Initial Clustering Threshold
    θ₂ : Cluster Merge Threshold
Output: Clusters of Constants
Method:
//Step i: Create an initial set of clusters
1:   Let CLUSTERS = {}
2:   For each constant v in CONSTANTS
3:       For each cluster c in CLUSTERS
4:           If (dist(v,c) < θ₁ ∧ v is closest to c)
5:               Add v to c and break
6:       If (v is not added to any c)
7:           Create a new cluster c' containing v
8:           Add c' to CLUSTERS
9:       Add c' to CLUSTERS
// Step ii: Cluster Refinement
10: Do
11:     For each cluster c in CLUSTERS
12:         For each constant v in c
13:             If ∃ c' ∈ CLUSTERS dist(v,c') < dist(v,c)
14:                 Reshuffle v to c'
15:     Recompute statistics per cluster in CLUSTER
16: While (a constant is reshuffled)
// Step iii: Cluster Merge
17: For each cluster c in CLUSTERS
18:     If ∃ c'∈ CLUSTERS. c'≠ c ∧ dist(c,c') < θ₂
19:         Merge c with c'
20: Output CLUSTERS
```

Figure 9: Constant clustering algorithm

$$strength(V, C) = \frac{\Sigma_{C_i \in L}.freq(V, C_i)}{Max_{L'!=L}.\Sigma_{C_i' \in L'}.freq(V, C_i')}$$

From the above formula, V is weakly associated to L if the constants in L are paired to V much less than constants in another cluster L' are paired to V. It is our intuition that the weaker the association between a variable V and L the greater is the likelihood that $\langle V, C, L \rangle$ is a real bug.

The number of clusters that are related to V. The number of clusters related to V is the number of different clusters that constants paired to V belong to. We denote this as $vDeg(V)$. It is our intuition that if a variable V is used with constants from many clusters, there is no or little restriction in pairing V with an arbitrary constant. Such a V might be "polymorphic" as it could be used with various "types" corresponding to various clusters of constants. Anomalies involving such a V are less likely to be real bugs.

Based on the above intuitions, we sort the candidate bug reports by the following formula:

$$suspiciousness(V, C, L) = \frac{size(L) \times log(size(L))}{strength(V, C) \times (vDeg(V) - 1)^2}$$

The larger the size of the cluster L, the more suspicious is the pairing. The weaker the association between V and C, the more suspicious is the pairing. Also, the more clusters V is related to, the less suspicious is the pairing. We give a higher weight to the size of the cluster L as compared to the association between V and C, by multiplying the size of the cluster L with the log of itself. Again, we give a higher weight to the number of clusters that V is related to (i.e., $vDeg(V)$) than the other two. To do so, we take the

square of $vDeg(V)$. As all bug reports intrinsically have a $vDeg(V)$ of at least 2, we subtract $vDeg(V)$ by one before taking the square. Thus $vDeg(V)$ does not contribute to the suspiciousness score when it is equal to 2.

At the end of the three steps, we report a candidate set of anomalies to be provided to the user for verification.

5. EVALUATION

In this section, we first present our experimental setting, followed by the results of applying our algorithm to the Linux 2.6.30 source code. We then consider some threats to the validity of our results and finally describe some uses of constants that were identified in analyzing our results, but that go beyond the scope of the current work

5.1 Experimental Setting

The analysis process was implemented using OCaml and carried out on a HP ProLiant server with two 3 GHz quad-core Xeon processors and 16 GB memory, of which only one core was used. The mining process was implemented using C#.Net 2.0 and carried out on an Intel Core2 Duo 2.40GHz 3.24GB RAM Windows XP Tablet PC. The total time for processing Linux 2.6.30 was around 4 hours, with all but a few minutes for the analysis.

We consider three strategies for the clustering process. In the "Pos" strategy, constants are only clustered by the position of their definition, *i.e.*, constants that are defined adjacently (no blank line between their definitions) are clustered together. In the "Pos&Beh" strategy, we first cluster according to position, and then refine the result using the clustering algorithm taking into account constant usage (behavior), as presented in Section 4. Finally, in the "Beh" strategy, we cluster only using behavior, as presented in Section 4. In all cases, reports are ranked, as described in Section 4. We consider only the top 20 reports in each list, as real bugs were quite sparse after this point in every case.

5.2 Results

The set of real and probable bugs obtained by applying our algorithm to the Linux 2.6.30 source code and manually analyzing the results is shown in Table 2. We consider a real bug to be one that has been fixed or acknowledged by the Linux developers, and a probable bug to be one that we believe to be a bug based on our study of the code.

The Pos strategy gives the worst results, with only a 25% precision among the top 20 results.[6] Next is Pos&Beh, with a 30% precision. Finally, is Beh, with a 50% precision. This suggests that position of definition is not a completely satisfactory indicator of the meaning of a constant, as some constants may be declared contiguously, but have a different purpose, while others may have definitions that are slightly separated, even though this placement might have no intrinsic meaning. The purely behavior based approach is sufficient to reconstruct the position relationships, when they are relevant, and to identify new relationships that are not made apparent by the positions alone. Finally, we observe that many of these bugs have been present for multiple years.[7] The long

lifetimes of these bugs suggests that this type of bug is not being identified by other approaches, whether automated tools or manual inspection.

The false positive with the highest suspiciousness score (i.e., 19,030, rank 5) when employing clustering by behavior only, is the bug report involving the constant `SUPPORTED_Pause` and the field `supported` of a structure of type `ethtool_cmd`. `SUPPORTED_Pause` is in a rather large cluster and has a low score for $strength(V, C)$, causing our approach to list it as a potential bug. In this case, there are two sets of constants that each have the same set of values. It sometimes occurs that the value of a variable containing one is copied into a variable containing the other, causing the constants to appear to be used with the wrong type of variable. The constant `SUPPORTED_Pause` is involved in a bit more such copies than most of the other constants with which it should be associated, and it is used a bit less often than those constants with the variables associated with its own cluster. This essentially represents a borderline case, and the algorithm unfortunately makes the wrong decision, putting this constant and two others with similar properties in the wrong cluster.

5.3 Threats to validity

As we verify our detected bugs with the Linux developers, we may assume that our identification of real bugs is accurate. Nevertheless, the Linux developers can only comment on results that they are asked about, and thus there is a danger of false negatives. In our approach, there are three primary sources of false negatives: parse errors, missing include files, and overly conservative manual evaluation of the results.

Our approach works primarily at the level of the abstract syntax tree, and thus it must be possible to parse the source code. As noted in Section 3, our parser does not apply the C preprocessor, and instead parses C preprocessor directives according to heuristics [18]. These heuristics, however, are not sufficient to parse some functions, and thus we are not able to collect information from them or find bugs in them. Additional heuristics could be added, however, we currently parse 97% of the Linux 2.6.30 code.

Our approach also relies on information contained in header files, to identify named constants and to obtain type information about nested structures. The Linux build process is complex, and thus we search for include files based on a few heuristics. Some include files, however, are missed, causing information about the associated constants and structure definitions to be overlooked. To address this problem, the parser could use path information found in makefiles, or use a wider default strategy for searching for header files.

Finally, we have manually analyzed the bug reports generated by our tool to identify those that seem like real bugs. This analysis may be too conservative, causing some real bugs to be considered as false positives. To address this problem, we could consult with Linux experts earlier in the report assessment process.

5.4 Other issues

In evaluating our results, we have identified two significant issues that are not targeted by the design of our algorithm and that may be beneficial to consider in future work. These are *subtyping*, in which a generic constant is associated with multiple clusters that also contain more specialized values, and *dependent typing*, in which the value of one constant

---

[6]We use the common measure of precision at k from information retrieval [14]. Recall is hard to measure in this case as the total number of variable-constant pairing bugs in the target program is not known.

[7]Linux versions are released roughly every 3 months.

Constant Name	Category	File Name	P	P&B	B	Lifetime	Status	Susp
TG3_FLG2_TSO_CAPABLE	context	drivers/net/tg3.c	no	no	yes	2.6.7-2.6.31	F	338,856
TG3_FLAG_10_100_ONLY	context	drivers/net/tg3.c	no	no	yes	2.6.15-2.6.33	F	276,454
VB_SISLVDS	context	drivers/video/sis/init301.c	yes	yes	yes	2.6.13-*	R	34,464
AHC_SCB_BTT	context	drivers/scsi/aic7xxx/aic7xxx osm.c	yes	yes	yes	2.6.0-*	R	31,068
EXT4_EXTENTS_FL	context	fs/ext4/inode.c	yes	yes	yes	2.6.27-2.6.32	F	12,723
USB_TYPE_MASK	value	drivers/net/wireless/zd1211rw/zd_usb.c	no	no	yes	2.6.19-2.6.31	F	2,957
NV_TX_VALID	name	drivers/net/forcedeth.c	no	no	yes	2.6.21-*	A	2,939
AHD_BUSFREEREV_BUG	context	drivers/scsi/aic7xxx/aic79xx pci.c	yes	yes	yes	2.6.16-*	A	2,844
ATH9K_INT_GLOBAL	value	drivers/net/wireless/ath/ath9k/mac.c	no	yes	yes	2.6.29-*	A	2,637
BIT_0	name	drivers/scsi/qla2xxx/qla mbx.c	yes	yes	yes	2.6.4-*	R	2,157

Legend: Context (see Section 1): 1) name = wrong constant name but with the right value, 2) value = wrong constant name, wrong value, 3), context = wrong context, right constant name. P = clustering by definition position only. P&B = clustering first by definition position, then by behavior. B = clustering by behavior only. * = a bug that has not been fixed in any release. Status: F = Fixed, R = Reported, A = Acknowledged. Susp = Suspiciousness score using B.

Table 2: List of bugs found

determines the cluster that should be used in some associated code context. We observe that the C++ language provides more strict type checking of enumeration constants than the C language. Nevertheless, neither subtyping nor dependent types are supported by enumeration types, and thus even in C++ it may be necessary to use some form of unsafe cast or unchecked named constants. The examples in this section are not restricted to bit-and and bit-or, or to structure fields as sources and sinks, and thus give a wider view of the problem than the one that is treated by our current approach.

Subtyping. Some of the constants used by Linux kernel code are masks that can be used to select a region of bits that then should be accessed using constants within a given cluster. Several clusters may share the same set of significant bit positions, and thus a single mask may be usable for these clusters. In this case, we may view the cluster containing the mask as a supertype of the clusters containing the specific values. Figure 10 shows an example. The function `tg3_get_5752_nvram_info`, at the top, tests for Flash 5752 values while the function `tg3_get_5761_nvram_info`, at the bottom, tests for Flash 5761 values. Other functions in the same file test for a mix of Flash 5752 and other values, suggesting a complex subtyping hierarchy where devices reuse some values and define some of their own. The mask `NVRAM_CFG1_5752VENDOR_MASK` covers all of these values, and thus can be used to extract the relevant bits in each case.

Another form of subtyping occurs when one file extends an existing cluster with new constants or gives new names to some values in an existing cluster. The constants `ADVERTISED_PAUSE` and `ADVERTISED_ASYM_PAUSE`, defined in the chelsio-specific header file `drivers/net/chelsio/common.h`, illustrate the latter case. These constants have the same values as the constants `ADVERTISED_Pause` and `ADVERTISED_Asym_Pause` defined in the more widely used header file `include/linux/ethtool.h`. A bug finding algorithm should allow the former constants to appear wherever the latter do.

Dependent typing. Dependent typing involving constants is most commonly found in function calls, where the value of one argument determines the cluster of another. It can, however, also occur for structure fields or variables. We also consider cases where the cluster of a structure field depends on the role that that instance of the structure plays, a situation that contradicts the hypotheses of the treatment of structure fields in our algorithm (Section 3).

```
switch (nvcfg1 & NVRAM_CFG1_5752VENDOR_MASK) {          1
 case FLASH_5752VENDOR_ATMEL_EEPROM_64KHZ:              2
 case FLASH_5752VENDOR_ATMEL_EEPROM_376KHZ:             3
  tp->nvram_jedecnum = JEDEC_ATMEL;                     4
  tp->tg3_flags |= TG3_FLAG_NVRAM_BUFFERED;             5
  break;                                                6
 case FLASH_5752VENDOR_ATMEL_FLASH_BUFFERED:            7
  . . .                                                 8
}                                                       9
```
drivers/net/tg3.c::tg3_get_5752_nvram_info::10278-10296

```
nvcfg1 &= NVRAM_CFG1_5752VENDOR_MASK;                   1
switch (nvcfg1) {                                       2
 case FLASH_5761VENDOR_ATMEL_ADB021D:                   3
 case FLASH_5761VENDOR_ATMEL_ADB041D:                   4
  . . .                                                 5
  break;                                                6
 case FLASH_5761VENDOR_ST_A_M45PE20:                    7
  . . .                                                 8
}                                                       9
```
drivers/net/tg3.c::tg3_get_5761_nvram_info::10435-10464

Figure 10: Illustration of mask-related subtyping

Figure 11, illustrates dependent typing at the function parameter level. In this code, the function `xm_write16` is called twice (lines 3 and 8), first with the third and fourth arguments as `XM_HW_CFG` and `XM_HW_GMII_MD`, respectively, and then with these arguments as `XM_RX_CMD` and a combination of constants of the form `XM_RX_*`, respectively. The constants `XM_HW_CFG` and `XM_RX_CMD` in the third argument are defined in the same enumerator type declaration, and thus can be considered to be likely to belong to the same cluster. On the other hand, `XM_HW_GMII_MD` is defined in a different enumerator type declaration than the `XM_RX_*` constants. Indeed, the value of the third argument determines the cluster of the fourth argument. This is a pattern that occurs often in low-level code that interacts with devices, but is out of the scope of the current approach.

Dependent typing also occurs, although less frequently, in the case of structure fields and variables. An example is shown in Figure 12, where the value in the field `RAP` determines the cluster of the value in the field `RDP`. Addressing this issue again goes beyond the current scope of our work.

The basic assumption of our treatment of structures, as presented in Section 3, is that for a given structure type, all instances of a given field are used in the same way. Some

```
if (hw−>phy_type != SK_PHY_XMAC) {        1
  ...                                      2
  xm_write16(hw, port, XM_HW_CFG, XM_HW_GMII_MD);   3
}                                          4
...                                        5
r = XM_RX_LENERR_OK | XM_RX_STRIP_FCS;     6
...                                        7
xm_write16(hw, port, XM_RX_CMD, r);        8
```

drivers/net/skge.c::genesis_mac_init::1581-1635

Figure 11: Function parameter dependent typing

```
lance−>RAP = CSR0;                         1
lance−>RDP = STOP;                         2
ariadne_init_ring(dev);                    3
                                           4
if (dev−>flags & IFF_PROMISC) {            5
  lance−>RAP = CSR15;                      6
  lance−>RDP = PROM;                       7
} else ...                                 8
```

drivers/net/ariadne.c::set_multicast_list::813-820

Figure 12: A dependently typed structure field

structures, however, violate this assumption. For example, the file `drivers/net/fealnx.c` uses a structure of type `fealnx_desc` to represent ring buffers used both in the transmission and reception of network packets. Transmission and reception buffers, however, are used in different ways, and thus a different set of constants is used in each case. Thus, some fields of a `fealnx_desc` structure can hold constants from different clusters, depending on the kind of buffer being represented. Most structures in Linux are, however, used only for a single purpose, and thus we have found that these kinds of false positives are uncommon.

6. RELATED WORK

A number of works use data mining to infer programming rules from common patterns in source code [5, 11, 17, 21, 23, 24, 25, 26, 28]. These approaches find rules such as "Whenever a set of program elements occurs in a method, another set of program elements must also occur in the method" [11] and "Whenever a method call is made, another method call must be made in the future" [26]. The work of Ramanathan et al. [21] includes some data flow analysis for identifying constraints on values. These studies have primarily focused on method calls, and in some cases can be generalized to other program elements such as structure fields [11]. However, none have considered the issue of bad constant-variable pairings, as done in this paper.

CP-Miner locates copy-pasted segments, and anomalies when one or a few copied segments are not modified like the rest [10]. It found 190,000 and 150,000 copy-and-pasted segments in Linux and FreeBSD, respectively. From the copy-and-pasted segments, 28 and 25 bugs were found in Linux and FreeBSD, respectively. While some bugs in constant usage may derive from copy-paste errors, our approach is not limited to that case.

MUVI correlates variable and structure field accesses, to find inconsistent update bugs, possibly related to concurrency [13]. Such accesses are explicit in the program, and thus they do not require a program analysis to collect information transmitted across local variables, as we have needed. They

also mine for association rules, describing a case where the presence of an object A implies the presence of another object B. Our clusters could be expressed as association rules involving disjunction, but obtaining such rules is out of the scope of the techniques used by MUVI.

Many clustering algorithms, such as k-means, k-medoids, etc. [6], takes as input the final number of clusters, which is not available in our setting. One could set this final number of clusters to be equal to the number of files or the number of variables paired with a constant, but either of these options is likely to produce poor clusters. Indeed, we have observed that a file can contain many clusters of constants, and we have observed that a constant in a cluster can be paired with multiple variables. Lo and Khoo extend k-medoids to be parameterless by incrementally increasing the number of clusters one-by-one and measuring the degree of cluster goodness [12]. The technique stops when a local optimum is reached. This technique is not suitable in our setting as the number of constant clusters is very large – i.e., there are many constant groups in the Linux kernel code and they are used for a wide variety of purposes. QT Clustering replaces the number of clusters with the maximum diameter of a cluster [7]. However, it requires m iterations, where m is the eventual number of clusters and this number is very large in our setting. X-means takes as input a lower bound (say, l) and an upper bound (say, u) of the number of clusters [20]. It extends k-means by initially creating l clusters and incrementally increasing it to u by splitting the intermediate clusters. The number of clusters resulting in the best degree of cluster goodness is reported. Muhr and Granitzer extend X-means by a merge operation to undo bad initial splitting of clusters [16]. As we do not know the precise range of the number of clusters, a wide range is potentially needed. Also, there are cases where both the original X-means algorithm and that of Muhr and Granitzer need more than $u − l$ iterations. The original X-means algorithm includes some optimizations based on a kd-tree data structure which could potentially be applicable to our algorithm as well. We could also investigate the effect of plugging our distance metric into the algorithms mentioned above and evaluate the quality of the clusters produced.

Sun et al. propose a data mining technique to find anomalies in a bi-partite graph [22]. Their approach uses random walk with restart. Unlike their approach, we detect anomalies using clustering. While the approach of Sun et al. does not take into account the weights in the links, we consider weights assigned to features used during clustering process. Also, we merge data mining with program analysis and show the utility of our hybrid technique in detecting real bugs in Linux kernel code.

The problem of inferring types for constants is an instance of the more general problem of inferring types from untyped data. Soft typing attempts to detect correctly typed terms in dynamic languages, and to insert the fewest possible number of run-time type checks at places where types cannot be inferred [2]. Soft typing, however, can rely on information about constants and operators that have only one possible type or only a few possible types, such as constants and arithmetic operations. No such information is available to our approach. In a problem closer to our own, Eidorff et al. use a type-based approach to identifying various representations of dates within COBOL code, to address the year-2000 problem [4]. They rely on annotations, that may be provided

by the user or inferred automatically based on rules defined in terms of common substrings. We briefly considered an approach based in part on common substrings for clustering constants, but found that it gave very poor results. Unlike the case of dates, we have no a priori knowledge of what kinds of strings might be relevant. Finally, work has been done on inferring types related to units of measure, but this work requires explicit type declarations [9].

Most of the bugs we have found are in device driver code. It is well known that such code is highly error prone [3], and one of the contributors to this is uncontrolled the use of constants. The Devil language addresses this in part by allowing the developer to declare the bit patterns occurring within constants in a high-level way [15]. Clay extends this approach with dependent types [27]. Devil and Clay are concerned with the values of constants, while our work is concerned with their names. Thus, the approaches are orthogonal.

7. CONCLUSION

Named constants are commonly used in systems code to denote various magic numbers and control options. Because compilers provide little or no type-checking support for these entities, developers can easily use inappropriate constants, with no warning from the compiler. This can lead to incorrect behavior and makes the code more difficult to understand.

We have proposed a new approach to capture bad variable-constant pairings via a hybrid program analysis and data mining technique. Our results show that our technique is scalable, as it can treat the entire Linux kernel within around 4 hours. For Linux 2.6.30, we have found 10 real or probable bugs. We have reported these bugs to the developers and a number of them have been subsequently fixed. In the future, we will consider operators other than bit-and, and bit-or, more types of sources and sinks, and the subtyping and dependent typing effects presented in Section 5.

8. REFERENCES

[1] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, Toronto, Canada, June 1991. ACM.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, Oct. 2001.

[4] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini in practice: A type-theoretic approach to the year 2000 problem. In *TLCA '99: the 4th International Conference on Typed Lambda Calculi and Applications*, number 1581 in Lecture Notes in Computer Science, pages 6–13, L'Aquila, Italy, Apr. 1999. Springer-Verlag.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, Oct. 2001.

[6] J. Han and M. Kamber. *Data Mining Concepts and Techniques, 2nd Eds*. Morgan Kaufmann, 2001.

[7] L. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: Identification and analysis of coexpressed genes. *Genome Research*, 9:1106–1115, 1999.

[8] International standard, programming languages – C, Aug. 2008. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1336.pdf.

[9] A. Kennedy. Formalizing an extensional semantics for units of measure. In *3rd ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM)*, Victoria, BC, Canada, Sept. 2008.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[11] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, Lisbon, Portugal, Sept. 2005.

[12] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, Portland, OR, USA, Nov. 2006.

[13] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 103–116, Stevenson, WA, USA, Oct. 2007.

[14] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[15] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *OSDI'00: the 4th conference on Symposium on Operating System Design & Implementation*, pages 17–30, San Diego, CA, USA, Oct. 2000. USENIX Association.

[16] M. Muhr and M. Granitzer. Automatic cluster number selection using a split and merge k-means approach. In *International Workshop on Text-Based Information Retrieval*, pages 363–367, Linz, Austria, 2009.

[17] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kohafi, and T. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, 2009.

[18] Y. Padioleau. Parsing C/C++ code without pre-processing. In *International Conference on Compiler Construction (CC'09)*, pages 109–125, York, UK, Mar. 2009.

[19] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[20] D. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Seventeenth International Conference on Machine Learning (ICML 2000)*, pages 727–734, Stanford, CA, USA, 2000.

[21] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Programming Language Design and Implementation (PLDI)*, pages 123–134, San Diego, CA, USA, June 2007.

[22] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *The Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 418–425, Houston, TX, USA, Nov. 2005.

[23] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Automated Software Engineering*, Auckland, New Zealand, Nov. 2009.

[24] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE*, 2009.

[25] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Automated Software Engineering*, pages 295–306, Auckland, New Zealand, Nov. 2009.

[26] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476, Edinburgh, UK, Apr. 2005.

[27] L. Wittie, C. Hawblitzel, and D. Pierret. Generating a statically-checkable device driver I/O interface. In *Automatic Program Generation for Embedded Systems*, Salzburg, Austria, Oct. 2007.

[28] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Automated Software Engineering*, pages 307–318, Auckland, New Zealand, Nov. 2009.