Static Analysis of Code Binaries for Safe Software Reuse

Ramakrishnan Venkitaraman and Gopal Gupta
Applied Logic, Programming Languages, and Systems Lab
The University of Texas at Dallas
Richardson, TX
ramakrishnan@student.utdallas.edu, gupta@utdallas.edu

ABSTRACT
In this paper we consider reusability of software component binaries. Reuse of code at the binary level is important because usually only the machine code for system components is available; vendors do not want to share their source code for proprietary reasons. We develop necessary and sufficient conditions for ensuring that software binaries are reusable and relate them to the coding standards that have been developed in the industry to ensure binary code reusability. These coding standards, in essence, discourage the (i) use of hard-coded pointers, and (ii) writing of non-reentrant code. Checking that binary code satisfies these standards/conditions, however, is undecidable, in general. We thus develop static analysis based methods for checking if a software binary satisfies these conditions. This static analysis rests on the abstract interpretation framework. We illustrate our approach by showing how we statically analyze the presence of hard coded pointer variables in assembly code obtained from binaries of digital signal processing applications. The analyzer we have developed takes the binary code to be checked for reuse as input, disassembles it, builds the flow graph, and statically analyzes the flow graph to check for the presence of code that will hinder its reuse. The construction of this analyzer is described and its performance results reported.

Keywords
Reuse of Software Binary Code, Static Analysis, Abstract Interpretation, Assembly Code

1. INTRODUCTION
Software components have received considerable attention in recent years. The dream is to develop a virtual market-place of commercial-off-the-shelf (COTS) software components developed by third party vendors. To assemble new applications, developers merely choose the right components and glue them together, perhaps with small amount of additional code (glue code). Software components thus promote software reuse (plug-and-play) which helps reduce software development time, development cost, and the time-to-market for new software based systems.

Most third party vendors are unwilling to provide the source code of the component due to proprietary reasons. Thus, in most cases, only binary code is available. Distributing software in binary form means that integration of the software with other applications does not require recompilation but only linking with the application. That is, application developers will use the API provided to call the functions available in the component, the application code will then have to be linked to the software components’ binary prior to being loaded in the main memory for execution.

However, the problem that arises then is ensuring that the software component is written in such a way that it does not hinder reuse of its binary. For example, the execution of component binary should not alter the application’s binary. The use of hard-coded pointers in assembly code obtained from binaries of digital signal processing applications is a condition that hinders reuse of these binaries.

Our work is motivated by practical concerns for software reuse in the digital signal processing (DSP) industry [2]. Texas Instruments (TI), world’s leading manufacturer of DSP hardware, is interested in developing a marketplace for DSP software COTS component. However, most of the DSP code from vendors is available as a binary for DSP processors in the TI TMS320 family. DSP software developers tend to use low level optimizations to make their software very efficient. One has to ensure that these low level optimizations do not interfere with reusability. Researchers at Texas Instruments have developed “general programming rules” as part of their Express DSP Algorithm Interoperability Standard (XDAIS) [12] that defines a set of requirements for DSP code (for TI TMS320 family of DSP processors). If DSP software developers follow this code, then it will be possible for system integrators to quickly assemble production quality DSP systems from one or more subsystems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXX-XX-X/XXX ...$5.00.

Note that software reusability is a very broad term [45], however, for our work we are primarily interested in reuse of software binary code. In the rest of the paper, the term software reuse should be taken to mean software binary code reuse.
Programming rules essentially lay down the restrictions that, if not followed, will result in code incompatibility during reuse (e.g., one rule restricts the usage of hard coded pointers in the program).

In this paper we analyze necessary and sufficient conditions for software binary code reusability. We relate these conditions to the TI’s XDAIS standard. The necessary and sufficient conditions are derived from the fact that linking and loading of binaries is done under certain assumptions. The binaries must not execute any instruction that will violate these assumptions.

Further, we are interested in developing automatic tools that will detect if a program is not compliant with these conditions. However, the compliance checking is undecidable in general (this constitutes the reason why TI has found it hard to develop tool for checking compliance of a program code with XDAIS standard [2]). We propose to use static analysis to perform this compliance check. However, static analysis of the program code is complicated by the fact that the source code of the program is not available—vendors generally just ship their binaries that can be linked with other codes. Thus, to check for compliance assembly code has to be analyzed. It should be noted that static analysis of assembly code is quite hard, as no type information is available. Thus, for example, distinguishing a pointer variable from a data value becomes quite difficult. Also, most compilers take instruction level parallelism and instruction pipelining provided by modern processors into account while generating code. This further exacerbates the automatic static analysis of assembly code. Our static analysis framework is based on abstract interpretation. Thus, assembly code is abstractly interpreted (taking instruction level parallelism and pipelining into account) to infer program properties. A backward analysis is used since in most cases data type of a memory location has to be inferred by how the value it stores is used. Once a point of use is determined, the analysis proceeds backwards to check for the desired property.

We illustrate our approach by considering how we statically analyze the presence of hard-coded pointers (rule 3 of the TI XDAIS [12] standard), i.e., how we check whether a pointer variable has been assigned a constant value by the programmer. We give details of the tool that we have built for this purpose. Other rules can be checked in a similar manner. Note that our goal in this project has been to produce an analyzer that can be used to check for compliance of large commercial quality program codes. Thus, all (nasty) features of C that may impact the analysis have been considered (most DSP code is written in C). For example, in hard-codedness analysis, we have to consider cases where pointers are implicitly obtained via array declarations, pointers with double or more levels of indirections (e.g., int **p), pointers to statically allocated global data area, etc.

Our work makes the following contributions:

- Necessary and sufficient conditions are developed for checking if a software binary code is reusable.
- A static analysis based framework is proposed for checking if these reusability conditions are satisfied. We show that software binaries can be successfully analyzed via abstract interpretation based techniques, even in the presence of instruction level parallelism and pipelining.

- The static analysis based framework is used to develop a practical system for compliance checking for one of the conditions (hard codedness of pointers).

We assume that the reader is familiar with abstract interpretation; tutorial introduction can be found in chapter 1 of [16].

2. REUSE OF SOFTWARE BINARIES

Third party code is generally made available by supplying a software binary that is linked with the applications that makes use of it. Vendors are generally reluctant to share their source code due to proprietary reasons. If software component binaries are to be reused, then one must ensure that these binaries do not include code that will compromise the reusability of the code. Reusability can be compromised, for example, if the code contains hard-coded pointers, or if the code is self-modifying or modifies other binaries that are linked to it.

Note that it is important to make a distinction between usability vs reusability of a software system. Certain, programming idioms, if used, may appear to compromise the reusability of the software system, however, on closer analysis they relate to issue of usability. For example, if a binary code makes array references that are out of bound, then this may appear as violating binary code reusability (the references may be to addresses outside the binary’s code or data area). Indeed, reusability is compromised if out of bound array references are made unintentionally. However, if out-of-bound array references are present unintentionally, then this means that the program still has software bugs that have not been removed. Thus, the software is not even usable, let alone reusable. In this paper we are not interested in issues of software usability.

Once binary code is given, it goes through two more steps prior to execution: (i) linking this binary with other binaries (done by a linker); and, (ii) the final loading of the resultant executable in the main memory by the loader after address relocation. A linker primarily performs the task of symbol resolution (determining relative offsets for labels), while a loader adds the proper offsets to addresses so that the program can be loaded in the area of the virtual memory allocated by the OS. Linking and loading operations are performed under certain assumptions. Reusability is compromised if the execution of this binary code results in these assumptions being violated.

A linker obviously assumes that the offset of a label will not change later. That is, given an instruction in a binary $B_1$ involving a label, e.g., `jump $L_1$, where $L_1$ is a label defined in another binary $B_2$ that is being linked to $B_1$, then the offset for $L_1$ should not change after linking. That is, the code starting from location $L_1$ will not be relocated somewhere else after the linking phase is over. Similarly, a loader assumes that an executable can be loaded anywhere in the virtual memory. Thus, to ensure reusability of binary code the following two conditions must hold:

- **C1** The binary code should not change during execution in a way that link-time symbol resolution will become invalid.
- **C2** The binary code should not be written in a way that it needs to be located starting from some fixed location in the virtual memory.
We assume that no additional information is given w.r.t. conditions under which a software component binary is to be used, apart from a specification of the API.

**Theorem:** Conditions C1 and C2 are necessary for binary code reusability.

**Proof:** The proof by contradiction is straightforward and is omitted.

These conditions are also sufficient, because they cover all the assumptions made during linking and loading. Note, however, that the necessary conditions above are hard to characterize and even harder to detect. Thus, in practice we broaden these conditions and consider more general conditions that are easier to characterize and detect. A broader condition that captures C1 is that the binary code should be re-entrant. Similarly, for C2 it is sufficient to check that there are no hard coded memory addresses in the program. Thus, checking for reusability can be reduced to checking for the following conditions: (i) C3: that the binary code is re-entrant; (ii) and, C4: the binary code does not contain any hard-wired memory addresses. Note that code re-entrance is a very useful way of characterizing conditions for reusability, because very often the same component binary may be executed by multiple threads or processes. Code re-entrance implies that such concurrent execution can take place safely.

**Theorem:** If conditions C3 and C4 hold, then the binary code is reusable (i.e., C3 and C4 are sufficient).

**Proof:** We will show that if C1 (resp. C2) does not hold then C3 (resp. C4) does not hold either. If C1 does not hold, then the symbol mapping for address labels determined at link-time does not hold at execution time. This implies that the symbol mapping was altered at execution time, i.e., the binary code got altered during execution, which in turn implies that the code is non-re-entrant. Similarly, if C2 does not hold, then there must be some address in the binary code that is used during the execution that is fixed. Thus, C4 does not hold.

Note that while C1 implies C3 and C2 implies C4, the implication does not necessarily hold in the other direction. Thus, the code may not be re-entrant yet may be reusable, as long as the modifications made to the binary at runtime are such that the symbol mapping is not altered and only one thread uses the binary code at any given time. Likewise, hard wired addresses may be present, yet the code may still be loaded anywhere as long as the specific hard wired addresses are known and they do not interfere with the area where the code is loaded. Thus, C3 and C4 are sufficient conditions, but not necessary conditions.

Finally, note that the application must be re-entrant as a whole. Checking for re-entrancy of a component binary may not be enough, because some other component binary may modify it during execution. Thus, each component binary when checked in isolation appears to be re-entrant, but when put together, it is not re-entrant.

### 2.1 The XDAIS Programming Standard

The coding standard rules, published by TI for software vendors of its DSP chips, that fall under the category of “general programming rules” [12] are the following:

1. All programs must follow the runtime conventions imposed by TI’s implementation of the C programming language.
2. All programs must be reentrant within a preemptive environment including time sliced preemption.
3. All data references must be fully relocatable (subject to alignment requirements). That is, there must be no “hard coded” data memory locations.
4. The code must be fully relocatable. That is, there can be no hard coded program memory locations.
5. Programs must characterize their ROM-ability; i.e., state whether they are ROM-able or not. ROM-ability means that if part of the executable is placed in the DSP ROM, it would still function; this restricts the way global data can be accessed (data cannot be placed in ROM) [12].
6. Programs must never directly access any peripheral device. This includes but is not limited to on-chip DMA’s, timers, I/O devices, and cache control registers.

Rule 1 is not really a programming rule, since it requires compliance with TI’s definition of C. However, rules 2 through 5 are manifestations of conditions C3 and C4 above. Thus, Rules 2 and 5 correspond to condition C3 while Rules 3, 4 and 6 correspond to condition C4. In light of these conditions, and examining the entire instruction set of TI’s TMS320 family of DSP processors, one can show that indeed the XDAIS standard is sufficient for ensuring that binary code that is compliant with it is reusable.

There are a number of advantages to DSP software vendors writing programs that comply with the published standards [12]. Compliance to standards (i) allows system integrators to easily migrate between TI DSP subsystems; (ii) enable host tools to simplify a system integrators tasks, including configuration, performance modeling, standard conformance, and debugging; (iii) subsystems from multiple software vendors can be integrated into a single system; (iv) programs are framework-agnostic, that is, they are reusable: the same program can be efficiently used in virtually any application or framework; and, (v) programs can be deployed in purely static as well as dynamic run-time environments (due to code relocatability).

### 2.2 Automatic Reusability Analysis

Next, we are interested in developing tools that automatically detect if a binary code is reusable. This entails automatically determining if any of the 5 programming rules above are not complied with. Detecting if rules 3, 4 or 6 are violated involves checking that there are no hard-coded references in the code. Checking for rules 2 and 5 involves ensuring that no writes are made to the code area during execution. Checking for hard codedness or checking that no writes are made to a specific memory area is undecidable [54] in general. Thus, one has to resort to approximating this automated checking. A standard method is to use static analysis [14]. Static analysis however is complicated by the fact that only binary code is available. All the type information is lost in the binary code, thus even determining if a value is an address or data is not quite that easy. In the rest of the paper we consider the problem of detecting hard-codedness and develop an abstract interpretation based framework for detecting hard-coded references. A similar analysis can be developed to check for code re-entrance (that is, for checking
that no writes are made to the code area; we do not discuss this any further due to lack of space.)

3. ANALYSIS OF HARD-CODED POINTERS

We illustrate our static analysis based approach to compliance checking by showing how we check compliance for rule #3, which states that there should be no hard-coded data memory locations. A data memory locations is hard coded in the assembly code if a constant is moved into a register $R_i$, and $R_i$ is then used as a base register in a later instruction. The constant value may of course be transferred to another register $R_j$ directly or indirectly, and then $R_j$ used later in dereferencing. Since most of the TI's DSP code is written in C, data memory locations can be hard coded either in assembly code embedded in a C program, or by using pointers provided in the C language. Thus, the problem of detecting hard coded references is to check whether a pointer variable that is assigned a constant value is dereferenced or not in the program. Thus, using the ‘C’ syntax for illustration purposes, given a variable $p$ of type `(int *)`, we want to check if there is an execution path between a statement of the type $p = k$, where $k$ is an expression that yields a constant value, and a later statement containing $*p$ (that dereferences a pointer). Of course, the dereferencing may take place directly or indirectly, i.e., we might have an intervening statement $(int *)q = p$ followed by a later statement containing $*q$.

Note that if a pointer variable is hard coded but never dereferenced, then the program is deemed safe. It is only after such a pointer variable is dereferenced during subsequent execution, that the program is deemed unsafe.

Clearly, the problem of detecting hard coded references is undecidable in general. So we employ static analysis for detection of hard codedness (from this point on, we’ll call the analysis hard-codedness analysis). The hard-codedness analysis analyzes each pointer variable and determines if the pointer is definitely hard-coded (HC), definitely not hard-coded (NHC), or that its hardcodedness status cannot be deduced. Thus, the analysis is conservative (as all static analyses are), in that there are instances where a determination of hardcodedness/non-hardcodedness cannot be precisely made. In practice, however, we have observed that our analysis—described in subsequent sections—is able to make a precise determination of hardcodedness/non-hardcodedness for nearly all pointer variables occurring in programs. Thus, in the benchmark programs we obtained from Texas Instruments, we rarely come across pointers whose hardcodedness or non-hardcodedness status could not be precisely determined (see Section 8).

Note also that the problem of detecting dereferencing of hard-coded pointers subsumes the problem of detecting dereferencing of NULL pointers. This is because a NULL pointer is a pointer that has been assigned a special constant (usually 0x0). Thus our analysis will also detect NULL pointer dereferences. Similarly, hard-codedness analysis subsumes analysis for checking if un-initialized pointer variables are dereferenced. This is because hard-codedness analysis attempts to check if a pointer dereference is reachable from a point of initialization; and thus will detect any pointers that are dereferenced but not initialized. Thus, our hard-codedness analysis performs two of the checks proposed by the UNO project [51] at the assembly level. The UNO project claims that NULL pointers, un-initialized pointers, and array out of bounds reference are three most common run-time programming errors.

A problem faced in statically analyzing binaries is that type information is unavailable, thus, distinguishing between constants stored in integer variables from constant addresses stored in pointer variables becomes difficult. The only way to distinguish between the two is to check if a register is dereferenced or used as a base register at some program point, and if so, we can go backwards from that program point and check to see if this register was directly or indirectly assigned a constant value.

We use an abstract interpretation [15, 16] based framework for static analysis. The abstract domain is quite simple and consists of four values: ⊥, ⊤, HC and NHC. Abstract operators are defined for pointer arithmetic based on this abstract domains, and the abstract values propagated in the flow-graph of the program (obtained by disassembling the machine code, and analyzing the control flow). The abstraction is shown to be safe (by constructing a Galois connection [15, 16]). Abstract semantics of the program are defined via recursive equations. The “collecting semantics” of the flow-graph is then computed via fix-points, which then allows us to check for hard-codedness.

A static analyzer has been implemented and used for analyzing a suite of DSP program codes obtained from Texas Instruments. Performance results on this suite of programs are reported. For most programs, our system is able to detect occurrences of hard-codedness with good accuracy. This is primarily because most practical DSP program codes use pointers in non-convoluted ways, and our static analyzer is able to determine the hard-codedness status in such cases with 100% precision.

4. ABSTRACT INTERPRETATION BASED STATIC PROGRAM ANALYSIS

Static program analysis (or static analysis for brevity) is defined as any analysis of a program carried out without completely executing the program. Static analysis provides significant benefits and is increasingly recognized [56] as a fundamental tool for analyzing programs. The traditional data-flow analysis found in compiler back-ends is an example of static analysis [14]. Another example of static analysis is abstract interpretation, in which a program’s data and operations are approximated and the program abstractly executed (abstraction is done in a way to ensure termination) to collect information [15, 16].

In abstract interpretation based static analysis, domains from which variables draw their values are approximated by abstract domains. The original domain is called a concrete domain. Further, for each operation over these domains, a corresponding abstract operation is defined over the abstract domain. A program is represented by a flow chart, and its semantics is given via a mapping from arcs that connect nodes of the flow chart to environments, where an environment is a mapping from variables to values in the concrete domain. In the abstract semantics, the environment is abstracted as a mapping from variables to values in the abstract domain. A state is defined as a pair $(arc, env)$ where $arc$ is an arc in the flow chart and $env$ is the environment that exists along that arc at a given moment. The same arc may be in different states (depending on the values of the variables), at
Our analysis is able to cope with this, since it regards execution depending on where the heap and stack are allocated. The meaning of the program $P$ is solution to the recursive equation:

$$P = \text{next} \cdot P$$

which is given by:

$$\text{fix}(\lambda f. \text{next} \cdot f)$$

In the abstract interpretation framework [15], a collecting semantics is used, i.e., we consider the set of all the abstract environments that might be associated with a program point (an arc in the flow-graph). This set of abstract environments is called a context. The collecting semantics thus associates a context with each arc. A context is a member of the powerset of the set of all environments.

$$\text{Contexts} = 2^{\text{Env}}$$

The context associated with a particular arc is the set of all environments that can exist along that arc, if we started execution from any of the initial nodes of the flow graph.

The collecting semantics is not computable because it gives exact information, it is therefore approximated. Given a pointer variable that is dereferenced, we are only interested in whether this pointer was hard coded earlier or not. Let $\mathcal{A}$ be the set of all memory addresses. An environment maps a pointer variable to an address in the set $\mathcal{A}$ (for hard codedness analysis we are only interested in pointer variables). $\mathcal{A}$ can be divided into two sets $\mathcal{A}_{\text{nhc}}$ and $\mathcal{A}_{\text{hc}}$, where $\mathcal{A}_{\text{nhc}}$ represents legitimate (i.e., not hard coded) memory addresses (those that might be returned by systems calls `malloc`, `calloc` or `realloc`, or returned by address-of operation, e.g., $\&$ operator in C), and $\mathcal{A}_{\text{hc}}$ represents the rest of memory addresses. Thus, $\mathcal{A}_{\text{nhc}} \cup \mathcal{A}_{\text{hc}} = \mathcal{A}$ and $\mathcal{A}_{\text{nhc}} \cap \mathcal{A}_{\text{hc}} = \emptyset$. We approximate the domain of addresses by an abstract domain, $\mathcal{A}_\alpha$, where $\mathcal{A}_\alpha = \{\bot, \mathcal{h}c, \mathcal{nh}c, \top\}$. Note that $\mathcal{h}c$ abstracts the elements in the set $\mathcal{A}_{\text{hc}}$ while $\mathcal{nh}c$ abstracts the elements of the set $\mathcal{A}_{\text{nhc}}$; the value $\bot$ represents complete lack of information, while $\top$ denotes that we cannot decide whether the value is $\mathcal{h}c$ or $\mathcal{nh}c$. $\mathcal{A}_\alpha$ forms a lattice as shown in Figure 1.

![Lattice Abstraction](image)

Note that we assume that NHC addresses are those that are derived from calls to memory allocation routines (`malloc`, etc.), and we also assume that any offset from an NHC address is also NHC. Thus, the sets $\mathcal{A}_{\text{nhc}}$ and $\mathcal{A}_{\text{nhc}}$ are determined by each program and may vary from execution to execution depending on where the heap and stack are allocated. Our analysis is able to cope with this, since it regards any address derived from `malloc`, `calloc`, `realloc` and the `&` operator to be safe, while any address directly assigned in the program is regarded as unsafe.

Following the abstract interpretation approach, we define the abstraction and the concretization functions, $\alpha$ and $\gamma$, respectively.

$$\alpha : \text{Contexts} \rightarrow \text{Abstract Contexts},$$

where $\text{Abstract Contexts}$ consists of abstract environments which map pointer variables to values in $\mathcal{A}_\alpha$:

$$\alpha(C) = \begin{cases} 
\bot, & C = \{\} \\
\mathcal{nh}c, & C \subseteq \mathcal{A}_{\text{nhc}}; \\
\mathcal{h}c, & C \subseteq \mathcal{A}_{\text{hc}}; \\
\top, & \text{otherwise}; 
\end{cases}$$

$$\gamma : \text{Abstract Contexts} \rightarrow \text{Contexts},$$

where

$$\gamma(S) = \begin{cases} 
\{\}, & S = \bot; \\
\mathcal{h}c, & S = \mathcal{h}c; \\
\mathcal{nh}c, & S = \mathcal{nh}c; \\
\mathcal{A}, & \text{otherwise}; 
\end{cases}$$

We next have to abstract the operators involving pointers (Table 1). Pointers can be involved in pointer arithmetic expressions that use $+$ and $-$ operators. Given a statement $p = q + i$ where $p$ and $q$ are pointers to integers and $i$ an integer, then if $q$ is hard-coded, our analysis should infer that $p$ is also hard-coded. Likewise, if $q$ is NHC, our analysis should infer that $p$ is also NHC. However, given pointers $p$, $q$, and $r$, and $p = q + r$, then for $p$ to be inferred as hard-coded, both $q$ and $r$ must be hard-coded. If, say, $q$ is hard coded but $r$ is not, then $q$ must be treated as an offset from the safe pointer $r$, and our analysis should infer that $p$ is not hard coded. With this in mind the definition of abstracted $+$ and $-$ operations for pointers is shown in the table below:

<table>
<thead>
<tr>
<th>+/−</th>
<th>hc</th>
<th>nhc</th>
<th>⊥</th>
<th>⊤</th>
</tr>
</thead>
<tbody>
<tr>
<td>hc</td>
<td>hc</td>
<td>nhc</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td>nhc</td>
<td>nhc</td>
<td>nhc</td>
<td>nhc</td>
<td>nhc</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>nhc</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊤</td>
<td>⊤</td>
<td>nhc</td>
<td>⊤</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Table 1: Pointer Arithmetic

Once the abstract operators are defined, we can compute the abstract semantics of the program by computing the fix-point of the recursive equation:

$$P_\alpha = \text{next} \cdot P_\alpha$$

where $P_\alpha$ is the abstract program (the abstract program ignores instructions that are determined to not affect a pointer value). Note that the abstract contexts (Abstract Contexts) forms a lattice under the subset relation, $\subseteq$, where the join ($\sqcup$) and meet ($\sqcap$) operations correspond to the set union and set intersection operations respectively. The join operation is used to merge the abstract contexts of multiple arcs leading into a common node. The abstract semantics once computed, tells us which pointer references are hard coded. Finally, we have to show that our analysis is sound. The soundness of the analysis follows if we can show that $\alpha$ and $\gamma$ are mutually consistent and that Abstract Contexts form a lattice.

Theorem: The Hard-codedness Analysis formulated above is sound.
Proof Sketch: Consistency of $\alpha$ and $\gamma$ functions is established by showing that they constitute a Galois connection [15]. That is:

$$x = \alpha(\gamma(x)) \quad \text{ ... (1)}$$

$$\gamma(\alpha(y)) \supseteq y \quad \text{ ... (2)}$$

It is easy to check that both (1) and (2) hold. Given the way Abstract_Contexts are defined, it is also easy to see that the set of Abstract_Contexts forms a lattice under the subset ($\subseteq$) relation. Thus, our analysis is indeed sound. □

5. THE ANALYSIS ALGORITHM

5.1 Overview

As discussed earlier, the analyzer has access only to the object (binary) code which is to be checked for compliance with the standard. So, the given object code is disassembled and the corresponding assembly language code is obtained. The disassembly is performed using TI Code Composer Studio [10, 11]. The disassembled code is provided as input to the static analyzer which produces a result which indicates whether the code is compliant with the rule.

To check for compliance with the standard, the binary code that is given as input is never executed. The analyzer scans through the disassembled code statically and checks whether there are any hard coded addresses. The basic aim of the analysis is to find a path from the point in which the dereferencing of a pointer occurs to the point at which an address is assigned to the pointer and then check whether that address is legitimate or not.

Figure 2 shows the various steps involved in the analyzer we have developed. After the disassembly of object code, the assembly code is split into functions. The analysis is first done function by function (corresponds to individual functions in the source code). For each function, the analyzer computes its basic blocks and constructs the flow graph. The flow graph is then statically analyzed. Once information for various functions (including main) has been collected, an interprocedural analysis is carried out, to analyze the entire program. The functions may have to be re-analyzed (until a fix-point is reached) during this interprocedural analysis.

It should be noted that there are advantages as well as disadvantages of performing static analysis at the assembly level. W.r.t. advantages, note that (i) parsing of source code is replaced by parsing of assembly instructions, which is considerably easier; (ii) in the source code pointers may occur in complex expressions (e.g., a pointer in a struct inside another struct), which makes the analysis complex at the source code level; at the assembly level all pointers manipulations are handled via registers, and thus all occurrences of pointers appear very similar, regardless of how they were expressed in the source code; (iii) since we analyze assembly code, which is source language independent, many of the nasty features of C get handled by the compiler and appear in considerably sanitized form in the assembly code.

W.r.t. disadvantages, (i) all type information is lost at the assembly level, so addresses are indistinguishable from data, thus abstracting information becomes harder, complicating the analysis. (ii) Registers are repeatedly reused, for holding values of different variables, and thus a lot of analysis is involved in figuring out which occurrences of a register in various instructions refer to the same value; (iii) Assembly code is much more verbose than source code, as well as it heavily employs pipelining and instruction level parallelism; thus, a very good understanding of the processor architecture is needed which needs to be then modeled in the abstract semantics. As an example, consider the following C code:

```c
void main()
{
  int a=1, b=2, c=3;
}
```

which translates to the assembly code shown below:

```
000007A0 07BE09C2 SUB.D2 SP,0x10,SP
000007A4 02002042 MVK.D2T2 B4,1,SP
000007A8 020012B2 MVK.S2 0x0002,B4
000007AC 023C22F6 || STW.D2T2 B4,++SP[0x1]
000007B0 02001A1B MVK.S2 0x0003,B4
000007B4 023C42F6 || STW.D2T2 B4,++SP[0x2]
000007B8 023C62F6 STW.D2T2 B4,++SP[0x3]
000007BC 00002000 NOP 2
000007CC 000C8362 BNOP.S2 B3,4
000007CC 00880052 ADDX.S2 B6,SP
000007CC 00000000 NOP
```

The presence of the $|$ characters denotes instruction level parallelism. We can see the usage of register B4 in line 5 and a value assigned to it in the line 4. But, the content of B4 used in line 5 is not the one assigned in line 4. This is due to the presence of parallelism. Note that our analyzer takes care of parallelism and instruction pipelining while computing the abstract semantics.

5.2 Phases in the Analysis

The analyzer functions in two phases. In the first phase, it scans through the flow-graph and detects all the register dereferencing that correspond to the dereferencing of pointer variables in the source code. It stores this information in the various basic blocks in a set. We call such a set a (potentially) unsafe set. The unsafe sets represent the abstract contexts discussed earlier. There is an unsafe set for each pointer that is dereferenced. This unsafe set records all the registers that may potentially hold the address corresponding to this pointer. In the second phase the unsafe sets are iteratively refined, until a fix-point is reached.

Phase 1: Detecting dereferencing of pointers: To detect the dereferencing of pointers, the analyzer starts from the entry nodes in the flow-graph and visits every reachable node in the flow-graph. While visiting any node in the flow-graph, it checks for the occurrences of pointer dereferencing. Dereferencing of a pointer is detected in the disassembled code when a register other than the stack pointer (SP) is used as the base register. Dereferenced registers are recorded in the unsafe sets. There is an unsafe set for each dereferencing operation in the program.

Phase 2: Checking if dereferencing is safe: In the second phase, the analyzer uses the information gathered in the first phase and ascertains the safety of each of the unsafe sets. This is done by analyzing the safety of the pointer across all possible paths through which the pointer might have got its value at the point of its dereferencing. That is, if there are multiple locations at which the same pointer may be hard coded (which may correspond to multiple paths in the flow-graph), the analyzer will be able to detect and report all such locations.
Refining Unsafe Sets: The unsafe sets are built iteratively, via a fix-point computation, as described earlier in the presentation of abstract interpretation framework. Initially, the unsafe set is empty; once phase 1 detects the dereferencing of a register, it adds that register to the set. So, if register “Reg” is seen as being dereferenced, it is added into the unsafe set, which will now appear as \{Reg\}.

In phase 2, the analyzer looks for statements in which an element from the unsafe set (in this case “Reg”) is used as the destination register. That is, the analyzer is trying to find what is the most recent value that was assigned to the register “Reg”. When it detects such an occurrence, say, which corresponds to a statement like “Reg = Reg1 + Reg2”, it deletes the element “Reg” from the unsafe set and inserts the elements “Reg1” and “Reg2” into the unsafe set. Now the unsafe set becomes \{Reg1,Reg2\}.

In phase 2, the analyzer continues with the current unsafe set (looking for the occurrence of both Reg1 and Reg2 as the destination registers in this case). Phase 2 terminates when the status of each of the unsafe set has been determined. As discussed earlier, the status of an unsafe set is deemed to be HC if all the elements in a given unsafe set are hard coded. If at least one of the elements in the unsafe set is not hard-coded, then the corresponding pointer is safe.

Note that if no pointers are dereferenced, the analyzer does not even enter phase 2. In phase 1, the analyzer inspects each line of the assembly code only once. This is achieved by maintaining a set of unsafe sets (SOUS) which is carried through as various instructions in the flow graph are examined.

Merging Information: During phase 2 of the analysis, the analyzer builds and populates the unsafe set. In the control flow graph, if a basic block has multiple successors, then the instructions of that basic block will be analyzed after merging the SOUS of all the successors. If the analyzer does not perform this merging operation then the analysis will be of exponential complexity. This is true especially if loops are also involved. The unsafe sets (abstract contexts) form a lattice under subset ordering, the merging involves a join (⊔) operation, which is simply a set union operation in this case. Static Analysis based approximation and merging of information makes the complexity of the analysis \(O(n \times m)\) where \(n\) is the measure of the size of the flow-graph, and \(m\) a measure of the size of (finite) lattices involved. Merging results in information loss. That is, since the common predecessor will be analyzed using a merged set, the information about the source paths of these merged SOUS are lost. But, merging information does not make the analyzer give incorrect results as the integrity of the individual unsafe sets is preserved.

6. HANDLING COMPLEX PROGRAMMING CONSTRUCTS

We next show how complex programming constructs are handled by our analyzer.

6.1 Handling Loops

A loop is detected when the predecessor of the current block is a block that has already been encountered in the analysis. Loops, if not properly handled, will result in the formation of wrong unsafe sets and incorrect results. So, as soon as the analyzer detects a loop, a new set called a loop-set, which is a set of set of unsafe sets, is created. That is, each element of the loop-set is a set of unsafe sets (thus, a loop-set is a set of SOUS). The analyzer also remembers the starting and the ending points (blocks) of the loop.

The first element added to the loop-set is the SOUS that the analyzer has computed when it detects the cycle. The analyzer then uses the current SOUS to analyze all the blocks forming the cycle including all the possible paths involving those blocks in the control flow graph. When the end block for the loop is reached, the analyzer checks if the current SOUS is already a member of the loop-set.

If the current SOUS is already a member, then the analyzer has reached a fixed point for the loop. That is, the analyzer has collected all the information from the loop and additional cycles through the loop will not add any new information to the sets. At this point the analyzer can safely exit the loop and continue the analysis of other unvisited blocks.

If the current SOUS is not already a member of the loop-set, then the analyzer merges (through the \(\sqcup\) operation in the lattice) the current SOUS and loop-set and continues to cycle through the loop with the current SOUS, until a fixed-point is reached.

Loops can be nested and in that case, the above procedure is performed for each of the inner loops. The fixed point is re-calculated for each of the inner loops for each cycle through the outer loop until the outer loop reaches a fixed point.

6.2 Handling Arrays

Operations on arrays, e.g.,

\[
\begin{align*}
... & \int a[] = \{\ldots\}; \\
... & a[.] = \ldots; \\
... 
\end{align*}
\]

exactly resemble pointer operations at the assembly level. That is, if statically allocated array elements are accessed, then the corresponding assembly code will resemble the dereferencing of pointer variables. The analyzer handles these cases by looking at what the destination location of the access corresponds to, i.e., whether it is in the stack or in the heap.

6.3 Pointers and Global Variables

There are two cases in which global variables influence the analysis. The first case arises when a pointer (local variable) is assigned a value from (say an integer after typecasting) a variable which is declared as global. The second case arises when a pointer which is declared as a global variable is dereferenced. In both cases interprocedural analysis is needed.
Consider that we have a pointer p dereferenced in a function and assigned a global variable. Since the variable is global, it could be modified by any part of the program. If there is no assignment to that global variable in the current function prior to dereferencing, then its value is coming from outside the function, and a global interprocedural analysis has to be performed. Similarly when a pointer, declared as global, is dereferenced, a similar situation arises. Interprocedural analysis is addressed in the next section.

6.4 Handling Functions

As already discussed, the analyzer first splits any given program into functions and analyzes each function for safety. However, inter-procedural analysis is needed to detect hard-coding in the following cases: (i) return values of functions may be hard coded pointers that are later dereferenced in the calling function; (ii) arguments can be passed as a reference to functions and the called function hard codes the arguments and the calling function uses the hard coded values; and (iii) function bodies have statements involving either a globally declared pointer or a pointer that is assigned an expression involving global variables.

We compute and keep track of the calling and return context for each function in a memo table. Thus, for each function, values are set in the memo table to denote whether the function arguments or return values are hard-coded. This memo table is built iteratively (since functions may be mutually recursive) until it reaches a fix point, during the analysis of the program. Similar iterative inter-procedural analysis is used in analyzing hard codedness of global pointers and pointers whose value depends on global variables.

Our analyzer pre-processes the flow graph to determine if some of the functions can be analyzed “stand alone.” These are analyzed and their resulting contexts stored in the memo-table in advance. Other functions are iteratively analyzed as outlined above.

6.5 Pointers requiring multilevel dereferencing

The usage of double pointers (**) or multilevel pointers that are hard coded complicates the analysis. This is because double pointers can be used to indirectly hard-cod other single level pointer variables.

When single level pointers (say int *p) is used for hard coding, the typical sequence of operations for hard-coding are as shown below:

```c
... somefunction(...) {
  declare pointer variables;
  hard code the pointer variables;
  create aliasing between the pointers;
  dereference the pointers;
}
```

Aliasing two pointers before hard coding one of them, will not affect the analysis, since after hard coding, the two pointers will no longer be aliased. However, in the case of doubly (or more) indirceted pointers there can be sequences such as the following.

```c
... somefunction(...) {
  declare pointer variables;
  create aliasing between the pointers;
  hard code the pointer variables;
  dereference the pointers;
}
```

A concrete example is shown below:

```c
void main() {
  int *p, val;
  int **q = &p;
  //p is hard-coded via q
  *q = (int*)0x8000;
  val = *p;
}
```

Detection of hard-coding in these indirect cases involves analyzing each line of the code multiple times or carrying huge sets of aliases, making the analysis very costly.

The analyzer that we have built flags a warning when it sees multilevel pointer dereferencing. Note that the analyzer will detect hard-coding for double pointers that are hard-coded directly, i.e., occurrences such as (int **p=90; **p; will be detected without any extra effort. Multilevel pointers cause problems in analysis only if we have these pointers used in hard-coding other pointers indirectly.

6.6 Handling Parallel Instructions

The || characters in the disassembled code signify that an instruction is to execute in parallel with the previous instruction[13].

```
istruction A
|| instruction B
|| instruction C
```

For example, if a code sequence as shown is encountered, where instructions A, B, C are some assembly level instructions then it means that instructions A, B and C are executed in parallel. That is, the instructions A, B and C in the fetch packet correspond to the same execute packet and are executed in the same cycle. Moreover, instructions A, B and C do not use any of the same functional units, cross paths, or other data path resources.

Static analysis of disassembled code needs to make sure that it handles such kind of parallelism. As soon as dereferencing of a base register or occurrence of an element in the unsafe set (as the destination register) is found to occur in parallel with other instructions, the analyzer continues analysis with the instructions that occur in the previous cycle for that register or matched element.

7. ILLUSTRATIVE EXAMPLES

In this section, we include some illustrative examples to show the capabilities of the analyzer developed.

Example 1:

```c
void main() {
  p = ...;
  q = 0;
  for(i=0;i<p;i++)
    q++;
  *q;
}
```
In example 1, q is a NULL pointer that is dereferenced after being modified in the ‘for’ loop. The analysis is able to detect that q is hard coded. Note that the analysis would have detected q to have dereferenced a null pointer if q had not been updated in the ‘for’ loop.

Example 2

void main()
{
    int *p, *q, i;
    q = malloc(sizeof(int));
    i = (int) q;
    p = (int*) i;
    *p;
}

In example 2, though p is assigned an int value, that value was derived from a safe pointer. Our analyzer will also detect p to be safe.

Example 3

void main()
{
    p = hard coded;
    q = good pointer;
    r = q + p - q;
    *r;
}

In example 3, though the dereferenced pointer r is derived as a function of a good pointer and a hard coded pointer, in reality we are assigning to r the hard coded pointer p. But since all pointer operations are abstracted, our static analyzer will not be able to detect this hard coding of r.

8. PERFORMANCE RESULTS

The performance figures for our analyzer are given in Table 2. The first column corresponds to the metrics that are used to gauge the performance of the analyzer, the subsequent columns show the performance for each of the selected DSP programs.

The programs that were used to test were taken randomly from the TI’s distribution of CCS[11]. We took the a.out files of the randomly selected programs, disassembled it, and then fed the assembly code to the analyzer. The analyzer produced a log file that contained the results of analysis and Table 2 was generated from the log file.

The value ‘Num fns’ corresponds to the number of functions in the program. ‘Num lines’ is the number of lines in the input file subject to analysis, while ‘Num BB’ is the number of basic blocks in the input file. ‘Num *Ptr’ is the number of pointers that were dereferenced in the file. ‘Num HC’ is number of hard coded occurrences in the program detected by the analyzer. ‘Max US size’ and ‘Avg US size’ are the maximum number and the average number of elements respectively in the unsafe sets created by the analyzer. ‘Max SOUS size’ and ‘Avg SOUS size’ are the maximum and average number of elements in the set of unsafe sets respectively. ‘Max Chain Len’ is the max path length from the point at which dereferencing occurs to the point at which the analyzer detects that the pointer is assigned a value. ‘RT (ms)’ is the running time of our analyzer in milliseconds.

From Table 2, we find that the average number of elements in the unsafe set at any point of time is small. One of the reasons is that modifications to the unsafe set always involve the deletion followed by the addition of one or more elements that got related to the deleted element. This also means that the number of elements to which a pointer gets related to through pointer arithmetic, assignment and other pointer operations is always small. Also, note that the number of element in the SOUS was a maximum of 13. This means that the maximum number of pointers that the analyzer was analyzing simultaneously is 13. The maximum chain length was 12 and in most cases the chain length is either 1 or 2. This means that most pointers were dereferenced immediately after getting hard coded. The amount of time that the analyzer takes to run depends on the number of hard coded pointers, the number of basic blocks in the disassembled code and the number of lines of code in the input file. The maximum time that the analyzer spent on analysis was less than 5 seconds on code with 1350 lines.

The results reported above were produced by the prototype implementation of the analyzer without any fine tuning. The main aim of building the current prototype was to prove the efficacy and viability of static analysis based tools to perform these checks in commercial software when source code is not available. While the largest program we have tried was only 1350 lines long, the sizes of the unsafe sets are relatively small causing the fix-point computation to converge rapidly. Thus, we are reasonably confident that our analyzer will scale up for larger code sizes (work is in progress to obtain larger benchmarks to test our system). Also, it should be noted that a component binary needs to be statically analyzed only once before being integrated into an application.

Finally, note that for each pointer variable in each of the benchmark programs, our analysis was able to precisely determine whether it is hardcoded or not hardcoded (i.e., none of the pointer variables were inferred by the analysis to have the value ⊥ or ⊤). Thus, we believe that a static analysis based approach will catch most cases of hardcodedness/non-hardcodedness in practical programs. Essentially, one has to write very convoluted programs for the analysis to infer the “don’t know” (⊤) abstract value for a pointer variable. Likewise, for the value undefined (⊥) to be inferred, the program must have undesirable features such as uninitialized pointer variables, unreachable code, etc. Most programs do not have such convoluted-structures/undesirable-features, especially if the creators of these programs have followed good software engineering and coding practices.

9. RELATED WORK

There is a wealth of literature on static program analysis. These static analyses either analyze data-flow or control-flow [14] of the program or employ an abstract interpretation based framework [15, 16]. However, much of this work has been done in a scenario where source code is available. Not as much attention has been paid to analyzing machine code. Several researchers have looked at link time optimization [17, 18, 19, 20, 21, 22], where machine code has to be analyzed, however, as pointed out by Debray et al [9], they are “limited to fairly simple local analysis.” There is a wealth of literature on pointer analysis [1, 3, 4, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 52,
since these analyses have to keep track of issues that arise are similar to the ones that arise in our analysis, access the same memory location or not. Many of the issues these efforts are concentrated on doing aliasing analysis of machine code, i.e., detecting whether two instructions will access the same memory location or not. Many of the issues that arise are similar to the ones that arise in our analysis, since these analyses have to keep track of use-def chains as well (i.e., given a use of a register, check where it was modified), however, hard-codedness analysis cannot be cast in terms of aliasing analysis.

In [9] Debray et al use a mod k abstraction in which no distinction is made between two addresses that have the same lower k bits to perform pointer aliasing analysis. An interprocedural, context-sensitive data flow analysis is performed to see if two instructions access the same abstract address. Fernandez and Espasa [7] extend this analysis a little further by also analyzing if a memory reference is to the heap, stack or global memory. Amme at al [6] perform a similar analysis to aid parallelizing compilers. They symbolically abstract the values of registers which are then propagated in the flow graph; dependence information is then gathered via data-flow analysis. Analysis of assembly code via data-flow analysis has also been used for other applications, these include estimating memory use and execution time for interrupt driven software [5] and verifying security properties [8, 43, 44, 46]

Static analysis has been recognized as an important technology for software quality assurance [50, 48], however, the limited efforts described in the literature primarily analyze the source code [49, 47, 48, 51, 50, 55]; none of them deal with code reusability. Those that analyze assembly code are only interested in security properties [8, 43, 44, 46] and not in reusability. Thus, to the best of our knowledge there is no existing work that statically analyzes assembly code to check for software reusability.

Other related work includes work done by software groups at Texas Instruments to develop tools for automatic compliance check. Static analysis based approach was considered too costly for the benefits obtained, and instead a testing-based approach was resorted to where a program code is run in different scenarios (for example, different parts of the memory) and checked to see if erroneous output is obtained. Of course, this method is not complete either. It also cannot pinpoint the problem (for example, the pointer that is hard coded cannot be automatically identified; all we can conclude that the code has some compliance problem).

Our results in this papers show, that contrary to belief, a static analysis based technique is effective, practical, and useful.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we developed and analyzed necessary and sufficient conditions for binary code reusability. We showed that absence of hard-coded memory addresses and code reentrance are sufficient conditions to ensure binary code reusability. However, automatically checking that these conditions hold for a binary code is undecidable in general. We proposed static analysis as a technique for approximating this check. We illustrate the approach by developing a static analyzer for analysis of hard-coded pointers, and develop an abstract interpretation based static analysis framework for this purpose. Our results show that static analysis based approaches are viable in industrial settings for checking for coding standards compliance. Code compliance checking is critical for code reuse and COTS compatibility in applications. A complete analyzer has been developed for pointer hard-codedness analysis and shown to run successfully on code samples taken from Texas Instruments’ DSP code suite. The prototype system is currently being refined to provide more accurate results in presence of global pointers and mutually recursive functions. We are also extending the system to handle rules 2, 4 through 6 [12] laid out by TI. The analysis needed for these rules is similar to that for hard-codedness and we are quite confident that a abstract interpretation based static analysis framework is sufficient.

Acknowledgments

We are grateful to Steve Blonstein of Texas Instruments for bringing the problem to our attention as well as answering various questions about TI’s DSP Chips and their instruction set.

11. REFERENCES


<table>
<thead>
<tr>
<th>Num ins</th>
<th>t_read</th>
<th>timerl</th>
<th>mcbpl</th>
<th>ligset</th>
<th>m_hdrv</th>
<th>dat</th>
<th>guiCodec</th>
<th>codec</th>
<th>stress</th>
<th>demo</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>126</td>
<td>196</td>
<td>292</td>
<td>345</td>
<td>950</td>
<td>1139</td>
<td>1188</td>
<td>1202</td>
<td>1350</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>15</td>
<td>30</td>
<td>22</td>
<td>72</td>
<td>48</td>
<td>49</td>
<td>28</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>0</td>
<td>19</td>
<td>6</td>
<td>10</td>
<td>102</td>
<td>109</td>
<td>105</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>40</td>
<td>28</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>1441</td>
<td>1270</td>
<td>1521</td>
<td>2262</td>
<td>2512</td>
<td>3063</td>
<td>3043</td>
<td>4505</td>
<td>4716</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Performance Results for the analyzer

53, 9, 6, 7, 8]. However, only a limited number of these [9, 6, 7, 8] consider analyzing machine code statically. Most of these efforts are concentrated on doing aliasing analysis of machine code, i.e., detecting whether two instructions will access the same memory location or not. Many of the issues that arise are similar to the ones that arise in our analysis, since these analyses have to keep track of use-def chains as well (i.e., given a use of a register, check where it was modified), however, hard-codedness analysis cannot be cast in terms of aliasing analysis.


[50] Improving software quality by static program analysis Horst Licheter and Gerhard Riedinger Proc. of SPI 97 software process improvement, Barcelona, 1997


[52] Demand-Driven Pointer Analysis Nevin Heintze, Olivier Tardieu Conference on Programming Language Design and Implementation 2001

[53] Points-to Analysis in Almost Linear Time (1996) Bjarne Steensgaard Symposium on Principles of Programming Languages


[55] George C. Necula, Scott McPeak, Westley Weimer. CCured Type-Safe Retrofitting of Legacy Code POPL ’02, Jan. 16-18, 2002 Portland, OR USA