An Evaluation of Free/Open Source Static Analysis Tools Applied to Embedded Software

Lucas Torri Guilherme Fachini Leonardo Steinfeld Vesmar Camara Luigi Carro Érika Cota
Universidade Federal do Rio Grande do Sul
PPGC - Instituto de Informática
Po Box 15064, ZIP 91501-970, Porto Alegre, RS, Brazil
{lucas.torri, gjafachini, lsteinfeld, vesmar.camara, carro, erika}@inf.ufrgs.br

Abstract

Static analysis can be a valuable strategy to improve the quality of embedded software at a lower development cost. In this paper, we have surveyed ten different free/open source tools that perform static software analysis and evaluated their use in embedded software. Experimental results show that the studied tools present widely different results, and most of them are not ready to be applied to embedded systems. Furthermore, we discuss possible directions to improve the use of static analysis tools in the embedded domain.

1. Introduction

Embedded systems assume day by day a more important position in global economy, with a worldwide market evaluated in 160 billion Euros and an annual growth of 9 percent [1]. Although embedded systems are more commonly associated to a hardware platform and hardware constraints, current systems are actually defined by the software whereas a single hardware can be used for a number of applications. This means that embedded software is no longer composed only of a few lines of assembly code, but can reach thousands of lines in assembly and high-level code, whose verification becomes a non-trivial task.

Differently than other kind of software though, embedded software has a stronger dependency with the hardware platform and development tools used. Even though high-level languages such as C or C++ have been largely used in this domain, the necessary proximity to the hardware requires the use of tools (compilers and associated development tools) that are tailored to specific processors or boards.

Despite these differences, software bugs in embedded systems are as or even more important matter than any other computer system, since failures in these systems can cause not only health risks, but also great money losses. According to [2], 80 percent of embedded systems fails are caused by the embedded software, and not the hardware. Therefore, testing embedded software is not only a very important task, but also a very expensive one.

Static code analysis is applied to detect defects early in the coding process and it focuses on typical errors that otherwise remain undetected during compilation. Thus, static analysis (SA) is applied after the code successfully compiles and before the start of code inspections and unit testing [3]. Since the cost of correcting bugs increases with the product development level, static analysis tools are an important aid for any software development process. For an embedded software, the most prominent advantage of SA is the ability of detecting faults that can compromise the reliability and safety of the embedded application, as reinforced in [3]. However, the potential reduction in the development costs and time-to-market are also very important reasons to include SA in the embedded software development flow.

Free and open source software (FOSS) is a widely adopted solution in software industry, providing quality de facto standards tools for developers, and helping to not aggravate projects costs [4]. This is not different with testing tools. Two good examples of FOSS testing tools are the ubiquitous unit testing framework for Java, JUnit, and the GUI testing tool, Selenium. Many automated static analysis FOSS tools have been developed in recent years to detect software anomalies such as dead code and unused data, security leaks, null pointer dereferencing, endless loops, and floating-point arithmetic problems [5-14].

Due to the advantages presented both by the static analysis tools and free/open source software, we have surveyed FOSS tools that perform static software analysis and evaluated their use in embedded software. Our main objective is to determine the quality of FOSS static analysis tools when applied, out of the box, to embedded software written in the C language, which is by far the most used language in embedded software development [1]. Although a few works have evaluated the use of static analysis tools [15,16], to the best of our knowledge, this is the first evaluation of such tools in the embedded software domain. Our experimental results show that such an evaluation brings an important contribution towards the definitions of a cost-effective verification strategy for embedded software.

The paper is organized as follows: Section 2 reviews the main concepts of static analysis and related works. Section 3 presents our experimental setup, including the tools and the used embedded applications. Section 4

978-1-4244-7785-2/10/$26.00 ©2010 IEEE

presents the results which are discussed in section 6. Section 5 concludes the paper.

2. Static Analysis and Related Work

Embedded software testing has been studied in the last few years and a number of approaches can be found [17-24]. Some dynamic testing strategies are proposed for specific platforms [17-20, 24]. Another line of research is the use of formal approaches, such as model-checking, for system verification [22, 26, 27]. Both approaches are of great value but suffer from the lack of flexibility and scalability.

Static analysis, on the other hand, has shown to be an effective tool for early fault detection at reasonable costs [25]. In the embedded systems domain, static analysis is normally associated to the verification of temporal properties of real-time systems [26, 27] or to code optimization [28]. In another application of SA, Reinbacher et. al. propose in [29] its use to support model checking of Intel MCS-51 microcontroller code by providing information that can be statically extracted from the source assembly code. Venkitamaran and Gupta [30] also analyze assembly code to automatically check whether code standards defined for a family of DSP processors have been followed by third party software developers.

A few authors have considered SA for fault detection in the embedded domain. Kowshik et.al. propose in [31] an annotation-based static analysis tool that verifies that critical components of an embedded control system do not depend on unmonitored values of other non-critical components. A critical component is the one that must ensure specific system conditions, but may share data with other system components. The SA tool requires an annotated C code (with restrictions on shared memory pointer usage) which is pre-processed before being analyzed. Chacko and Jacob [32] present a code validation tool that analyses machine-level code against a set of rules based on the instruction set and architectural features of a particular processor. The rules are defined after a thorough analysis of various instructions used for configuring the integrated peripherals of the target processor. Despite its effectiveness, one important disadvantage of this technique is its dependency on the set of rules that must be defined for each target processor, which constantly changes in an embedded development process.

Considering the challenges associated to the application of dynamic test strategies for embedded software, and the need for a test method less tied to a specific platform and more adaptable to available development frameworks, we evaluate in this paper the suitability of available FOSS static analysis tools to the embedded software domain. Zitser et. al. [15] have performed a similar evaluation but considering non-embedded open-source code (specifically network-related applications). The main goal of our study is to determine whether traditional analysis and tools are sufficient to deal with the specificities of the embedded software. Furthermore, we determine possible improvements that can help to increase the coverage of static analysis in the embedded domain. Our experimental results show that this type of analysis must still be improved to deal with the specificities of embedded software without loosing generality and flexibility.

3. Experimental Setup

3.1 Open Source/Free Static Analysis Tools

The Internet is a vast repository of FOSS static analysis tools. There are plenty of them available through it. We have experimented several of those, and selected the most prominent ones and those we did not find problems to run.

In this paper, we have surveyed ten different free/open source SA tools and evaluated their use in embedded system's software. Our main objective is to determine the quality of these analysis tools when applied, out of the box, to embedded software written in the C language, which is by far the most used language in embedded software development. A short description of the tools used in this work is presented below:

- ***GCC*** [5]: the C Compiler from the GNU Compiler Collection is available for a huge number of embedded platforms. It may sound a little bit strange that the GCC compiler is on this list, but in fact it is able to warn of some code problems that can be statically discovered. We used it as a reference, to check the kind of problems a common compiler would be able to detect.
- ***CBM*** [6]: is a Bounded Model Checker for ANSI-C and C++ codes. It claims to verify buffer overflow, pointer safety, exceptions and user-specified assertions. It is also aimed for embedded software and supports dynamic memory allocation using malloc and new.
- ***Splint*** [7]: is a well-known static analysis tool and the newer tool from the "lint" family. It checks C programs for security vulnerabilities and programming mistakes. According to Splint's manual, the problems the tool can detect are: dereferencing a possibly null pointer, using possibly undefined storage or returning storage that is not properly defined, type mismatches, with greater precision and flexibility than provided by C compilers, violations of information hiding, memory management errors including uses of dangling references and memory leaks, dangerous aliasing, modifications and global variable uses that are inconsistent with specified interfaces, problematic control flow such as likely infinite loops, fall through cases or incomplete switches, and suspicious statements, buffer overflow

vulnerabilities, dangerous macro implementations or invocations, and violations of customized naming conventions.

• ***RATS*** [8]: Rough Auditing Tool for Security, is a tool for scanning C, C++, Perl, PHP and Python source code and flagging common security related programming errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions.

• mygcc [9]: mygcc is an extensible version of GCC, that can be easily customized by adding user-defined checks for detecting, for example, memory leaks, unreleased locks, or null pointer dereferences. User-defined checks are performed in addition to normal compilation, and may result in additional warning messages. GCC already includes many built-in checks such as uninitialized variables, undeclared functions, format string inspection, etc. Mygcc allows programmers to add their own checks that take into account syntax, control flow, and data flow information.

• Yasca [10]: according to its documentation, Yasca helps software developers ensuring that applications are designed and developed to meet the highest quality standards. On practical words it is related to quality assurance testing and vulnerability scanning. Nevertheless, it does not specify which problems it is able to catch.

• UNO [11]: its main goal is to intercept the three most common types of software defects: use of uninitialized variable, null-pointer references, and out-of-bounds array indexing. Also, it allows the specification and checking of a broad range of user-defined properties that can extend the checking power of the tool in an application driven way. Properties can be specified, by writing simple C-functions, for instance, for checking lock order disciplines, compliance with user-defined interrupt masking rules, rules stipulating that all memory allocated must be freed, etc.

• Flawfinder [12]: a program, written in python, that examines source code and reports possible security weaknesses ("flaws"') sorted by risk level. It works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, potential shell metacharacter dangers and poor random number acquisition.

• Sparse [13]: Sparse provides a set of annotations designed to convey semantic information about types, such as which address space pointers point to, or which locks a function acquires or releases.

• Cppcheck [14]: it claims to do scope check, bound checking, check of deprecated functions, memory leaks, redundant if, bad usage of the function strtol, bad usage of the function sprintf (overlapping data), division by zero, unsigned division, unused struct member, passing parameter by value, check how signed char variables are used, condition that is always true/false, unusual pointer arithmetic, dereferencing a null pointer, and incomplete statement.

3.2 Applications under Test

To evaluate the previous listed tools, we used five different applications written in C:

1. Mixed Code;
2. Traffic Lights;
3. Arduino's project bootloader;
4. Darjeeling Virtual Machine;
5. Dalvik Virtual Machine.

The purpose of the first application is to evaluate how many errors and exactly which errors each tool can discover. Therefore, we gathered several examples provided by the tools themselves and united them as a single test program. This application presents no particular embedded system characteristic, but there are a total of 17 errors on it. On the other hand, case studies 2 and 3 are two well tested embedded applications, and known to be correctly working. We have not introduced any kind of flaw on them, because our objective with these two examples was to see how the tools would behave when applied to code that is intended to run in embedded platforms. The Traffic Lights is an application capable of controlling cars and pedestrian traffic lights. It was built using the "device driver" philosophy, where hardware dependent parts are isolated from the main logic of the application. Its functionality is driven by an infinite loop, which is a common characteristic in embedded system's applications. Arduino is an open source embedded prototyping platform [33]. The platform uses a bootloader to simplify onboard applications loading, which we used as a test application. It is a very hardware dependent application, and includes inline assembly onto its code. Finally, the last two case studies are implementations of virtual machines. Darjeeling is a Java Virtual Machine (JVM) aimed to embedded systems [34]. It was designed for use in 8 and 16 bit microcontroller platforms. In order to achieve this, several features from the Java language were dropped, making it possible to run meaningful programs in as little as 2kB, where other JVMs often require at least several hundreds kB of RAM. On the other hand, Dalvik is the core runtime of the Google's Android operating system and it is optimized to run on low resources computing systems, such as smartphones and netbooks [35]. The build process of Dalvik VM is automated by the Android build system who is based on the make tool and on a set of scripts who configure the environment to the build (in Darjeeling this is done by an ant script). Such a build approach is not unusual for embedded systems, where platform-specific code is located in a file hierarchy to make it easier the constant changes in the target platform. Both Dalvik and Darjeeling have a similar proposal of transforming Java's bytecode into an optimized bytecode, in order to execute them in an embedded environment.

Each application can be successfully compiled under the x86, ARM, or AVR versions of the C compiler of the GNU Compiler Collection.

4. Experimental Results

The first three applications were checked one by one through the static analyzers. We note that the software libraries implementations used by the applications were not tested, as they are supposed to be reliable due to the extensive use and support. All SA tools, together with documentation, were fairly simple to use. All of them work through a command line interface, receiving the list of files to analyze as parameters.

Since each tool has a distinct range of target faults, we considered a common set of faults composed of the six more common problems addressed by this kind of tools according to [3] and one extra problem that is addressed by most of the selected tools. The target faults are classified in seven types: division by zero (DV), memory leak (ML), null pointer dereferences (NP), uninitialized variable (UV), buffer overflow (BO), inappropriate cast (IC), and local variable pointer return (LV). The local variable return error occurs when a pointer to a local variable is used as a return of a function as shown in the code of Figure 1.

```
char
*function_returns_pointer_to_sta
ck() {
    char foo[10];
    return foo;
}
```

Figure 1: Example of a local variable return error

4.1 Evaluation for the Non-embedded Application

Table 1 exhibits the number of errors of each type the tools were able to discover in the Mixed Code test suite. The last line indicates the total number of errors existed for each kind of fault in that application. The two errors found by GCC where pointed out when the *Wall* flag was set. The other tools were run several times, using different flags, and the total amount of errors among all executions is presented in Table 1. Some of the tools were configured by external files. With RATS we used its default database and for mygcc we used a check file available at the tool´s website.

From Table 1, one can observe that the different tools have a completely different range of detected problems. Many tools were not able to pinpoint errors they were supposed to detect. This is the case, for instance, of Flawfinder and RATS, which did not discover any buffer overflow. Indeed, both are designed to point security weaknesses and they issue a message every time a fixed-size buffer is found in the code. The message only indicates a possible source of error and is issued even if the code is actually correct. Splint proved to be the most effective tool, evidencing 13 out of 17 known errors. Notice that RATS and Flawfinder were not able to find any fault.

Besides the defined subset of faults, some tools were able to detect unused variables (which we did not consider as a fault), and also gave guidelines for good programming practices. Also, Flawfinder classifies the source code being analyzed in levels of risk. Strangely, it gave some of the lowest risk level to the examples application despite the three buffer overflow erros present in the code.

Table 1 – Results for the Mixed Code Example

	DZ	ML	NP	UV	BO	IC	LV
GCC						1	1
CBMC	1						
Splint		3	4	3		2	1
RATS							
mygcc		3	1			1	1
Yasca		1					
UNO				2	2		
Flawfinder							
Sparse						1	1
Cppcheck		3					1
# faults in the code	**1**	**3**	**4**	**3**	**3**	**2**	**1**

All tools but Yasca display their results, by default, in the system's command line. Yasca is the only one that supplies reports as an HTML file, being by far the most polished and well finished one. Cppcheck deserves a mention as well, because their results are displayed in a very simple, but effective way. Reports of most tools, on the other hand, are a little bit confusing, merging together the results of the analysis and the messages derived from the tool processing steps.

Thus, the variability of the results presented by different AS tools reported in [15] has also been observed in our case study. Furthermore, the excessive number of false positives often precludes the careful analysis of the really important directives provided by the SA tools.

4.2 Evaluation for Embedded Applications

When SA tools are fed the embedded applications, a successful analysis may require code changes, in order to avoid some of the syntax pitfalls presented by the tools. Tools CBMC and Splint presented problems. Splint was not able to recognize the *asm* keyword, used for inline assembly code in GCC. In order to correctly analyze the bootloader application, we had to comment blocks were the keyword was used.

Most flaws reported for the embedded applications were false positives. For example, the infinite loop of the Traffic Lights application and also a second loop that was controlled by a flag changed by an interrupt were pointed out by Splint and CBMC as errors. With CBMC we had to comment several lines that, despite correct, were not recognized by the tool due to syntax problems. For the bootloader application, this problem was such that the suitable modified code turned into a completely different application, thus invalidating the analysis.

Since Splint was the one with the best results for both, the test code and the first two embedded applications, we proceed with the analysis of the virtual machines using only Splint. Still, in both cases, the tool was not able to analyze the whole code at once, forcing the division of the program into small chunks of code. The analysis of Dalvik virtual machine was specially challenging because of the automated build process. The Android automated build process predefines some environment variables which defines the right "include" files that will be compiled with the application code. Those "include" files are also inspected (although not analyzed) by the Splint tool. In Darjeeling, we analyzed each module independently, as the analysis in the complete source code failed (Splint crashed).

Similarly to what was observed in Section 4.1, the results of the analysis contained a considerable amount of irrelevant warning messages, which are not considered as real flaws. For both VMs, deliberate coding decisions made by developers and common in this type of code, were pointed out as errors, even though the code is correct. On the other hand, the tool did not detected an actual memory leak problem in Darjeeling VM (the problem was detected by manual inspection).

5. Discussion

Static analysis can play a major role in the design of high-quality embedded software with reduced impact in development and test costs as well as in time-to-market. The use of FOSS SA tools is, in its turn, an interesting approach to reduce the dependence on specific platforms and to help the design of portable code, which further reduces costs and time-to-market.

However, from the results presented in Section 5, it seems clear that available FOSS SA tools do not suffice to deal with embedded software. Indeed, the tool that presented the best results in terms of number and type of faults detected (Splint), still required numerous modifications in the source code to analyze typical embedded software. An important issue, specially important for embedded software, for instance, is the need to understand and deal with automatic build process, mainly of code that is distributed and can be changed in a regular basis. Furthermore, the elevated number of false positives together with additional misleading results, preclude a careful manual analysis over important and real error indications. This happens because the programmer easily gets bored by the repetition of the same message over again in the output, thus loosing interest and completely bypassing similar remaining messages.

Thus, despite the intrinsic limitations of the static analysis, we believe that the adaptation of an available FOSS SA tool for the embedded domain can not only increase the detection rate, but also improve the quality of the analysis by focusing on more specific problems. For instance, it seems from the results that analysis is performed by the tools in a very local manner whereas the analysis of a certain depth in the code can possibly be considered to at least reduce the number of false negatives. FOSS tools are specially prone to these adaptations.

For example, a kind of checking that is not done by the current static analysis tools and could be interesting to embedded system are memory boundaries checking. In C embedded software it is pretty common to declare pointers to memory positions, like in the piece of code shown in Figure 2.

```
void
change_some_memory_position() {
  int *p = (int *) 0x500000;
  *p = 5;
}
```

Figure 2: Example of inexistent pointer address

However, if the hardware platform does not have that specific address, the compiler would still compile the code without any warnings.

Similarly, the C standard defines size ranges to all language variables. This means that, depending on the used compiler, a short could have the same size in bits of an int. In the tested analysis tools, cast from two variables of those types would be classified as an error, what actually would be a false positive.

In both cases, a basic configuration file with some basic information about the target platform would suffice. In this sense, since mygcc and RATS are able to receive different configurations, are worth a second evaluation.

6. Final Remarks

This paper discussed the role and applicability of static analysis FOSS tools into embedded software. As a conclusion, we suggest two possible adaptations to currently open-source SA tools to address those problems. Current work includes the evaluation of tools RATS and mygcc with distinct configuration faults. We´re also considering additional embedded applications to define exactly the range of faults (specific to embedded systems) a static tool can catch in an embedded code. Finally, we consider the inclusion of this verification approach in a complete and cost-effective test methodology for embedded software.

7. References

[1] C. Ebert e C. Jones, “Embedded Software: Facts, Figures, and Future”, *Computer*, vol. 42, 2009, pp. 42-52.

[2] Y.K. Jooyoung Seo, “Which Spot Should I Test for Effective Embedded Software Testing?,” 2nd. Intl. Conference on Integration and Reliability Improvement, Jul. 2008.

[3] D. Brook & Metrowerks, "Improving Embedded Software Test Effectiveness in Automotive Applications", *Embedded Systems Europe*, vol. 8, no. 55, pp. 16-17, February 2004.

[3] Chelf,B; Ebert,C. "Ensuring the Integrity of Embedded Software with Static Code Analysis", *IEEE Software*, vol. 26, no. 3, pp. 96-99, May/June, 2009.

[4] D. A. Wheeler, "Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!". http://www.dwheeler.com/oss_fs_why.html, 2007 [Accessed Oct. 7, 2009]

[5] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)". http://gcc.gnu.org/. [Accessed: Oct. 7, 2009].

[6] "The CBMC Homepage". http://www.cprover.org/cbmc/. [Accessed: Oct. 7, 2009].

[7] "Splint Home Page". http://www.splint.org/. [Accessed: Oct. 7, 2009].

[8] "RATS - Rough Auditing Tool for Security" http://www.fortify.com/security-resources/rats.jsp. [Accessed: Oct. 7, 2009].

[9] "mygcc". http://mygcc.free.fr/. [Accessed: Oct. 7, 2009].

[10] "Yasca - Yet Another Source Code Analyzer" http://www.yasca.org/. [Accessed: Oct. 7, 2009].

[11] "Uno Tool Synopsis" http://spinroot.com/uno/. [Accessed: Oct. 7, 2009].

[12] "Flawfinder Home Page" http://www.dwheeler.com/flawfinder/. [Accessed: Oct. 7, 2009].

[13] "Sparse - a Semantic Parser for C ". http://www.kernel.org/pub/software/devel/sparse/. [Accessed: Oct. 7, 2009].

[14] "SourceForge.net: cppcheck". http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page. [Accessed: Oct. 7, 2009].

[15] Zitser, M.; Lippmann, R.; Leek, T. “Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code”. ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2004, pp. 97-106.

[16] Wedyan,F.; Alrmuny,D. and Bieman, James M. “The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction”. International Conference on Software Testing Verification and Validation, 2009, pp.141-150.

[17] Sung, A.; Choi, B.; Shin, S. “An interface test model for hardware-dependent software and embedded OS API of the embedded system,” *Comput. Stand. Interfaces*, vol. 29, 2007, pp. 430-443.

[18] Seo, J.; Sung, A.; Choi, B; Kang, S. , "Automating Embedded Software Testing on an Emulated Target Board," Intl. Workshop Automation of Software Test , 2007. pp.9-9, 20-26 May 2007

[19] Wu, X.; Li, J.; Lee, W. D. and Lee, Y. “Coverage-Based Testing on Embedded Systems.” 2nd Intl Workshop on Automation of Software Test. 2007.

[20] Guan, J.; Offutt, J.; Ammann, P. “An Industrial Case Study of Structural Testing Applied to Safety-critical Embedded Software”. ISESE’06, September 21–22, 2006, Rio de Janeiro, Brazil.

[21] Okika, J. C.; Liu, Z; Ravn, A.P.; Siddalingaiah, L. ”Developing a TTCN3 Test Harness for Legacy Software”. AST’06, May 23, 2006, Shanghai, China.

[22] Pfaller, C.; Fleischmann, A.; Hartmann, J.; Rappl, M.; Rittmann, S.; Wild, D.” On the Integration of Design and Test - A Model-Based Approach for Embedded Systems” AST’06, May 23, 2006, Shanghai, China.

[23] Quynh, B.Thi and Aktouf, Oum-El-Kheir. “Diagnosis Service for Embedded Software Component based Systems”, EFTS'07, September 4, 2007, Dubrovnik, Croati.

[24]Yu, R. ”Fiscal Cash Register Embedded System Test with Scenario Pattern”, *International Journal of Computer Science and Network Security*, Vol.6 No.5A, May 2006

[25] Zheng, J.; Williams, L.; Nagappan, N.; Snipes,W.; Hudepohl, J.P.; Vouk, M.A. “On the Value of Static Analysis for Fault Detection in Software”, *IEEE Transactions on Software Engineering*, Vol. 32, No. 4, April 2006.

[26] Lettnin, D; Nalla, P.K.; Ruf, J.; Kropf, T.; Rosenstiel, W. ”Verification of Temporal Properties in Automotive Embedded Software”. IEEE Design, Automation and Test 2006.

[27] Chen,K.; Malik,S.; August, D.I. “Retargetable Static Timing Analysis for Embedded Software”, ISSS’01, October 1-3, 2001,pp.39-44.

[28] Regehr, J.; Reid, A. “HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems”, ASPLOS’04, October 9–13, 2004, pp.133-143.

[29] Reinbacher, T.; Brauer, J.; Horauer, M.; Schlicht, B. “Refining Assembly Code Static Analysis for the Intel MCS-51 Microcontroller”, SIES, 2009, pp.161-170.

[30] Venkitaraman, R.; Gupta, G.. “Static Program Analysis of Embedded Executable Assembly Code”, CASES, 2004, pp.157-166.

[31] Kowshik, S.; Rosu, G.; Sha, L., "Static Analysis to Enforce Safe Value Flow in Embedded Control Systems", Intl. Conf. on Dependable Systems and Networks, 2006, pp.23-34, 25-28 June 2006.

[32] Chacko, M.; Jacob, P., "Validation of Embedded Software through Static Analysis of Machine Codes," IEEe Intl. Conference on Advance Computing, pp.1596-1601, 6-7 March 2009.

[33] "Arduino - HomePage". http://www.arduino.cc/. [Accessed: Oct. 7, 2009].

[34] "Darjeeling - Java for micro controllers". http://darjeeling.sourceforge.net/. [Accessed: Oct. 7, 2009].

[35] "Android | Official Website". http://www.android.com/. [Accessed: Oct. 7, 2009].