Testing ThumbPod: Softcore Bugs are Hard to Find

P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, S. Yang, A. Hodjat, B. Lai, I. Verbauwhede
EmSec, Electrical Engineering Department, UCLA
Los Angeles, CA

ABSTRACT

We present the debug and test strategies used in the ThumbPod system for Embedded Fingerprint Authentication. ThumbPod uses multiple levels of programming (Java, C and hardware) with a hierarchy of programmable architectures (KVM on top of a SPARC core on top of an FPGA). The ThumbPod project teamed up seven graduate students in the concurrent development and verification of all these programming layers. We pay special attention to the strengths and weaknesses of our bottom-up testing approach.

1. INTRODUCTION

ThumbPod [1] is an embedded fingerprint identification system. It allows to capture and process the latent fingerprint image of a person, and extract a unique signature in the form of minutia data. Next, the minutia data are compared to a stored template. Embedded in the form factor of a keychain, ThumbPod is the equivalent of an electronic, biometrically secure key. Communication with the outside world proceeds through the use of a security protocol that avoids direct transmission of biometric data.

The operation of ThumbPod is complex and requires cooperation of many different design elements. Fingerprint minutia detection and matching is a complex image processing problem. The security aspects in ThumbPod require the use of encryption and hashing algorithms. And the embedded, battery-operated context requires this to proceed in a power- and performance-efficient manner.

Our design flow uses a divide-and-conquer strategy that considers the ThumbPod design at multiple levels of abstraction as shown in Figure 1. For system integration and security protocol design, we use Java programming. Contemporary Java technology provides a smooth translation to embedded context by means of the K Virtual Machine (KVM) [2]. For low-level programming, as well as for integration of software IPs, we use C programming on top of an embedded SPARC Processor. Finally, we also make use of dedicated VHDL coding to customize the Sparc processor with encryption- and signal-processing coprocessors. The current prototype runs on top of a XC2V1000 FPGA with 32MB of RAM.

The ThumbPod project (`http://www.thumbpod.com`) was run over eight months with a group of seven graduate

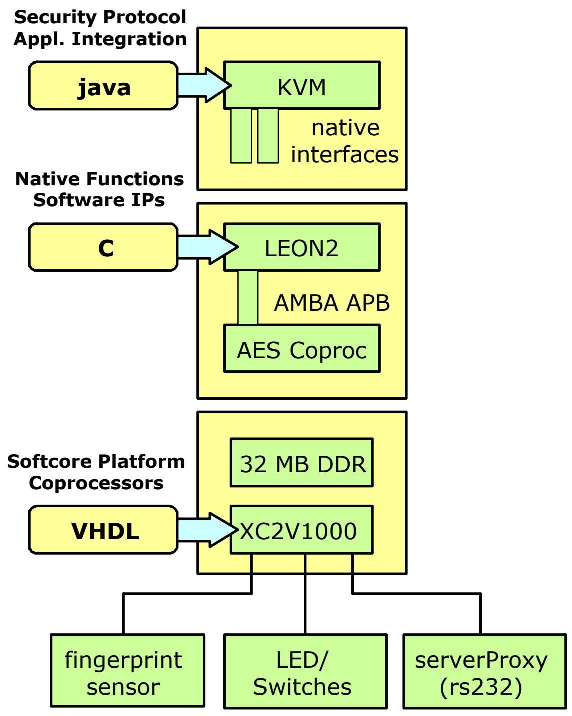


Figure 1: Programming Levels in ThumbPod

students, who were working in small teams at different programming abstraction levels. While the project started out in an informal academic setting, the complexity of the verification and validation process required us to introduce more rigorous techniques, including strict versioning, automatic regression testing and daily code build.

In this paper, we first review related work. Next, we discuss the setup of the ThumbPod design flow and show what verifications are done at each abstraction level. Following this we highlight the difficulties we face in the design and verification process, and enumerate specific techniques that we used to alleviate those. Not all of them are solved, and therefore we can also point out a few open issues in the conclusions.

2. RELATED WORK

Currently many fingerprint verification systems are com-

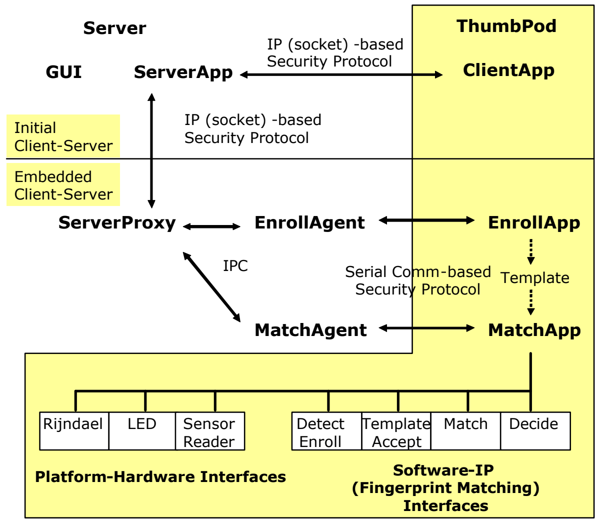


Figure 2: ThumbPod Application System Architecture

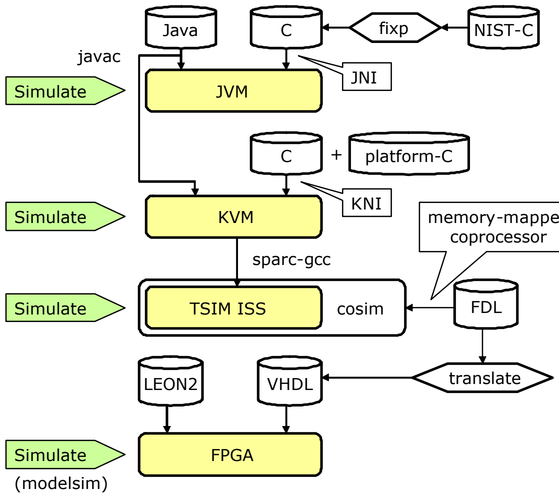


Figure 3: Design and Verification Flow, with codesign interfaces in call-out.

mercially available [3]. Very few, however, are applicable to a deeply embedded context, where the fingerprint sensor is integrated *together* with biometrics processing into a portable battery-operated device. Yet this configuration is very relevant since it is the only one in which the fingerprint biometrics are — by definition — free from eavesdropping [4].

The use of hierarchically programmable systems with softcores in FPGA's, for example, sheds new light on the verification process. Traditionally, abstraction layers in an embedded system have been strictly separated, making the implicit assumption of a sequential development model in which lower levels are correct and stable [5]. In ThumbPod, multiple levels of programming are used as a means to master design complexity and increasing performance. However, one cannot assume the implicit correctness of the neighbouring programming layers, as all of them participate at the same time in the design process.

3. THE THUMBPOD DESIGN

3.1. Application Architecture

In this paragraph, we look at the system-level context of ThumbPod, considering the operating environment of ThumbPod. The ThumbPod application architecture is illustrated in Figure 2. The ThumbPod keychain is configured as a client to a server that requires authentication of a ThumbPod user. This server could for example be located at a bank institution. The client-server communication is designed as a security protocol on top of the Internet IP protocol. The security protocol avoids the transmission of raw biometric data by using a challenge/response mechanism. The server will send a challenge to the ThumbPod that can only be answered by means of a sucessful fingerprint authentication. For this purpose, ThumbPod securely stores two reference templates that corresponds to the true user. Each authentication is done against both templates. As a result, we increase the accuracy of the authentication process.

In embedded context, the ThumbPod client decomposes into several hardware and software processes. A `ServerProxy`, `EnrollAgent` and `MatchAgent` are used as an intermediate between the ThumbPod keychain and the server. The keychain contains two application processes, `EnrollApp` and `MatchApp`, that are used for user enrollement and authentication respectively. The agent applications are running on a terminal that is located at the place of authentication. For example, in the secure payment scenario this can be a merchant terminal. The ThumbPod keychain connects to this terminal by means of a serial communication link.

Although the ThumbPod keychain implements only `EnrollApp` and `MatchApp`, it is important to consider the overall system architecture. Indeed, the system level testplan must take into account that ThumbPod is only the client of a complete client-server application.

The system level model of ThumbPod was written in Java. The use of Java is an initial design decision, motivated by the natural support that Java offers for networked and security applications. As a consequence of this the Java Virtual Machine model has to be adapted to the embedded ThumbPod context. Specifically we introduce native interfaces to make platform-specific features available as Java methods. By careful design, this binding can be done transparantly from the application programmer [1].

Our native interfaces fall in two categories. A first set is defined by the integration of embedded hardware platform features. This includes hardware coprocessors, fingerprint

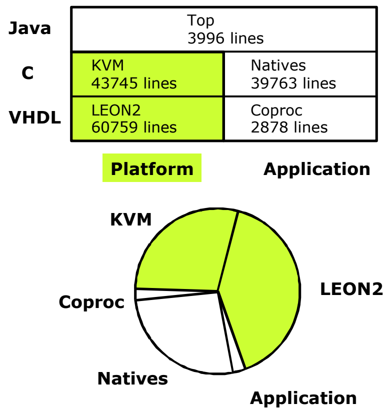


Figure 4: Code Complexity of ThumbPod

sensor and low-level I/O elements (LEDs and switches). A second set of interfaces is defined by the integration of fingerprint matching software, which was developed in C. The native interface concept makes it easy to separate component-level testing from system-level testing. On the other hand, it also introduces extra design challenges, because the development of native interfaces and their implementation proceeds concurrently with system level developments. A systematic design flow, introduced in the next section, is used to combine all design levels together.

3.2. Programming Levels and IP reuse

Thumbpod has three layers of programming: Java, C and VHDL. IP reuse was essential at each abstraction level. The design and verification flow is illustrated in Figure 3. The client-side of the ThumbPod application is originally written as a high-level functional model in Java and runs on top of JVM. The fingerprint identification subsystem is build starting from NIST fingerprint detection code in C [6]. Using signal-processing design techniques, this code is adapted for embedded operation with fixed-point precision. The customized C routines are integrated into JVM using the Java Native Interface (JNI). The high-level ThumbPod model is then ported to an embedded Java Virtual Machine, KVM. At this level, also other platform-specific native functions are added. For example, networking, I/O operations and Java class loading all become platform-dependent in embedded context. The resulting KVM port is verified on a workstation. Next, the KVM is also ported to the embedded LEON-2 processor, including the integration of specific interfaces for the fingerprint sensor and the coprocessors. We use an instruction-set simulator to verify the LEON-2 cross-compilation and the crypto- and signal-procesing co-processors. For this purpose we make use of an internally developed cosimulation tool [7] that interfaces cycle-true models of the coprocessors to the instruction-set simulator. Finally, the LEON-2 HDL is ported, together with a translated version of the coprocessors, to an FPGA board

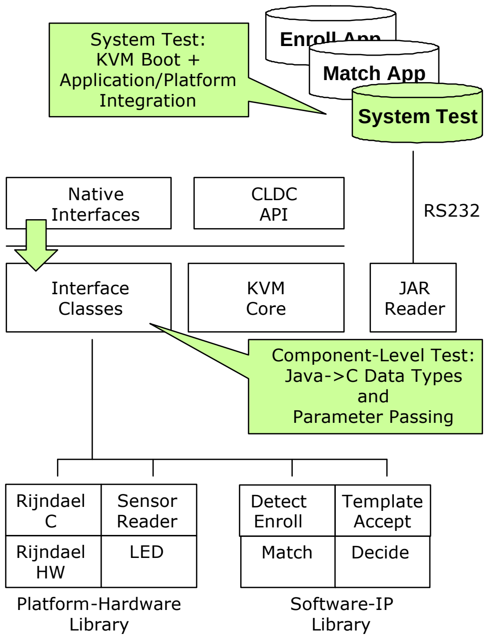


Figure 5: KVM Architecture

In this design flow, there are four independent, concurrent development efforts. A first team concentrates on the top-level Java code, with specific attention to the development of the security protocol. A second team is working on the embedded fingerprint matching software in C. A third team takes care of the KVM platform and native interfaces from Java to hardware. And a fourth team is developing the hardware platform, including the copocessors and sensor interfaces. Each of those teams is working towards a common goal and deadline. Interfacing the development efforts of all those teams, and integrating their design results, is a major challenge in the ThumbPod project.

3.3. Code Complexity

Figure 4 shows the absolute and relative complexities of different components ot the ThumbPod client, taking the amount of (non-commented) source code as a first-order metric. The three programming levels of ThumbPod are shown. The *Top* level contains Java. The second and third levels consist partly of *platform code*, and partly of *application-specific code*. Platform code introduces additional levels of programming abstraction, and includes KVM and LEON-2. Application-specific code implements ThumbPod-specific services, and includes for example the fingerprint detection native functions in C and the coprocessor models in hardware.

Considering the relative complexities, we find that only 31% of the code base is application specific, yet in order to

Java Native Method

```
public class InsightLeonMatch {
    public native void match(byte[] template1);
}
```

C Implementation of Java Native Method

```
JNIEXPORT jint JNICALL Java_insight_
          InsightLeonMatch_match(JNIEnv *jenv, jobject job,
                                 jbyteArray array1) {

  u8 array1_c[2800];
  jbyte *array1_pointer;
  jsize array1_len;

  array1_len      = (*jenv)->GetArrayLength(jenv,array1);
  array1_pointer  = (*jenv)->GetByteArrayElements(jenv,array1,0);
  if (array1_len != 2800) {
    fprintf(stderr, "Error - parm 1 length match() must be 2800\n");
  }
  ...
}
```

Figure 6: Java Native Method Parameter Passing

get the application to work in an FPGA, *all* code must be integrated, compiled and verified. We estimate that 31% in our project is even a relative high number due to the large amount of fingerprint-related C code which is application specific. If we would leave out the fingerprint-related C code, then only 5% of the ThumbPod code would be application-specific.

4. TESTING THUMBPOD

Our first approach towards constructing a design and test plan for ThumbPod is based on a bottom-up strategy that uses point-tests for each abstraction level individually.

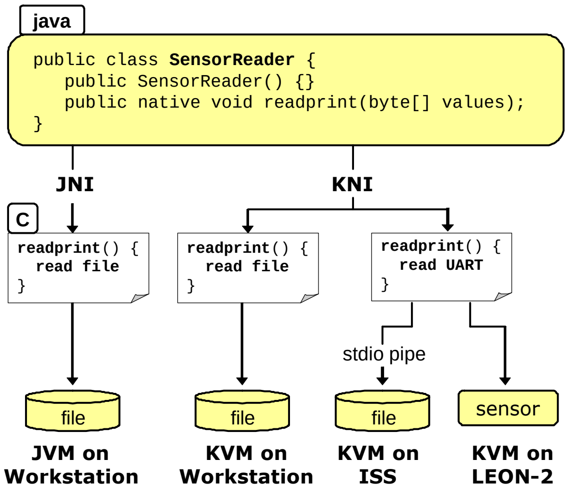
4.1. Java-Level Functional Test

At the Java level, we use functional simulation wich abstracts out all implementation constraints. Such a simulation tests various aspects of the communication protocol that ThumbPod uses to address the server-side of the application. This includes the detection of security problems in the matching protocol (false fingerprints, replay attacks, false thumbpod and/or server identity) as well as the detection of functional problems (timeouts and transmission errors).

4.2. Embedded Java Functional Test

While the functional tests aim at verifying the overall client-server protocol, a second suite of tests consider the embedded Java context on top of KVM. The architecture of the embedded KVM running the ThumbPod client is shown in Figure 5. KVM defines a number of programming API known as profiles. We use a stripped down version of the basic CLDC (Connected Limited Device Configuration) profile. Customizations of the KVM are done in two areas: A custom JAR reader is developed to bootstrap Java applications onto KVM, and all native interfaces are integrated into the KVM API.

At this level, each individual native interface is tested separately using a small standalone program. One point of



```
public class SensorReader {
   public SensorReader() {}
   public native void readprint(byte[] values);
}
```

```
readprint() {
  read file
}
```

```
readprint() {
  read file
}
```

```
readprint() {
  read UART
}
```

Figure 7: Example of native implementation of fingerprint sensor under multiple simulation contexts

particular attention is the transition from Java to C. While there is a standard API in KVM to support data transfer from Java to C, the user remains entirely responsible for the transfer process. Consequently, coding of native interfaces becomes highly error prone. An example of a native method invocation in Java and the corresponding C implementation is shown in Figure 6.

Table 1: Main Native Interfaces of ThumbPod

Platform-Hardware Native Interfaces	
Rijndael	Encryption/ Decryption using AES, either as a HW coprocessor or in SW (C)
SensorReader	Interface to fingeprint sensor reader or filesystem, returns a fingerprint image
LED	Low level IO (switches & LEDs)
Software-IP Native Interfaces (fingerprint matching SW)	
DetectEnroll	Accepts a fingerprint image and returns fingerprint minutia
TemplateAccept	Accepts two sets of minutia and decides if they are valid as a template
Match	Accepts two sets of minutia and returns their matching score
Decide	Accepts two matching scores and decides wether the ThumbPod user is valid or not

The use of multiple simulation platforms also requires testing of different aspects of native function design. In Figure 7, the example of reading the fingerprint sensor is shown. The top of the figure shows the Java class `SensorReader`, which contains a native function `readprint` through which fingerprint data can be read into the virtual machine. For simulations on a workstation, the actual sensor is substituted with the filesystem. Both a

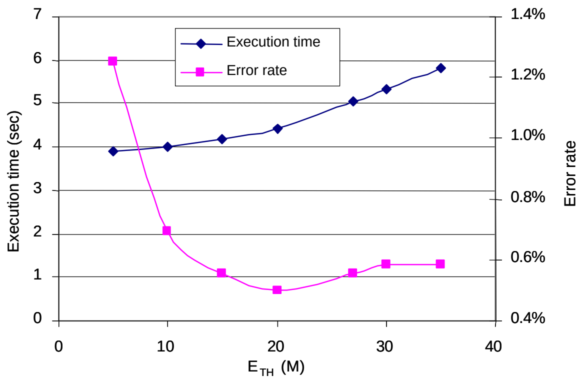


Figure 8: Dependency of Fingerprint Minutia Detection Quality and Algorithm Runtime

JVM- as well as a KVM-version are required. As pointed out the native interface mechanism of the standard JVM (JNI) is not the same as that of the embedded KVM (KNI). For the embedded processor target, we substitute the file operation with a UART I/O operation. When KVM runs on LEON-2, the sensor is accessed. But when this function is used on top of the LEON-2 instruction-set simulator (ISS), the UART I/O operation maps to standard I/O and ultimately to a file. This way, an appropriate set of simulator configurations is determined for each of the native interfaces in ThumbPod, which are enumerated in Table 1.

4.3. Fingerprint Authentication Test

.A major component of ThumbPod are the Fingerprint Detection and Matching routines. They are developed and tested separately. The goal is to map the algorithms to embedded, fixed-point (32-bit) context with good performance. The acceptance criteria for fingerprint matching are *false-accept-rate* (FAR) and *false-reject-rate* (FRR), which correspond to acceptance of a false identity and rejection of a true identity respectively. FAR is considered to be a more critical parameter, but there is a tradoff between optimization for good FAR and good FRR [8]. We gradually collected a database of 100 fingerprints over the project. This data is cross-matched in exhaustive overnight simulations to establish the FAR and FRR after each improvement of the algorithms. This way we achieved commercial-grade 0.01% FAR and 0.5% FRR.

As with many signal processing algorithms, there is no single 'good' solution to quality. A particular feature is that execution time and matching quality turn out to be iexchangeable parameters. Figure 8 gives the example of the execution time and detection quality of the minutia detection algorithms, which are dependent on a convergence threshold parameter E_{TH}. The existence of such factors motivates a continous optimization of the algorithms throughout the project. But as a result, it is no longer possible to say when a native method is really 'finished'

4.4. Hardware Test

Finally, the hardware platform is simulated at RT-level to verify the coprocessor interfaces. The simulation speed of the HDL simulator (Modelsim XE) is in the order of ten LEON-2 instructions per minute and too slow to consider extensive simulations, or even to boot a complete C program. We therefore use small boot programs that perform dedicated tests of the coprocessor interfaces. In the end, we also found that it is easier to debug the application by-implementation on the FPGA. The synthesis traject of our design, which fills 80% of a XC2V1000, has a smaller turn-around time than the RT simulation traject.

4.5. Showstoppers

Despite the design and test at multiple levels of abstraction, we failed a crucial intermediate deadline for 'version 1.0' of ThumbPod. Upon investigation it turned out that our bottom-up testing strategy was insufficient. This strategy focuses on the correct operation of individual components (Java application, KVM, Fingerprint C code, Hardware), but not on the project result itself. A lot of problems show up upon integration of individual components, as illustrated by these examples.

- While the interface between the Java application code, and the Fingerprint detection code is compatible, the actual data representation exchanged over these interfaces is not.
- The fingerprint image quality of the *actual* hardware sensor is in some cases insufficient for the Fingerprint Algorithms.
- A KVM application that works well on an ISS becomes unreliable on the actual hardware due a memory hardware problem.
- The codebase of the project shows a lot of redundancy. Sharing of code is done at source code level, and multiple copies of the same code co-exist. This is mostly a result of incremental, bottom-up developments in a research context.

We conclude that our testing strategy should focus more on the entire project, and have a top-down aspect. The code base is reorganized, and a number of techniques are introduced to master the integration complexity. We enumerate those in the next section.

4.6. Project Techniques

- *No Source Sharing:* All sharing between the different components (Java/C/KVM/Hardware) is done at the binary level through the use of class libraries and object code.
- *Build Automation:* Each component is organized in a separate directory with automatic make facility. Each make has four targets: `compile`, `release`, `test`, and `clean`.
- *Daily Build:* Each day, the entire project is compiled, tested and released during an overnight compile. The error log of this procedure is distributed each day by email.
- *CVS Version Control*: While the use of CVS is introduced early-on in the project, we find that the usage policy

of version control (tags and timely commit) is at least as important as the versioning itself.

- *Code Review*: By explaining the organization and structure of code to a peer, some obvious cross-component interface problems were identified.
- *Target Dates instead of Starting Dates*: Our planning is adapted to indicated the target dates of activities rather then the starting date. This way, dependencies between components were made explicit.

Figure 9 shows a bug curve over time, obtained by examining the reports from the *daily build* activity. The original slipped v1.0 deadline was week 19 of the project. From week 20 to week 23, the techniques indicated above were gradually introduced. The major release points in the project are indicated as well. While 'zero bugs' does by no means imply that ThumbPod is error-free, it *does* imply that the complete ThumbPod is error free to the extent of our tests.

5. OPEN ISSUES

In retrospect we consider which testing techniques would have allowed us to get ThumbPod right first-time. Given the heterogeneous mix of technology that is used in this project, clearly a lot of problems show up at the fringes, in this case the interfaces. For example:

- The use of Java as a strongly typed language is great, but the advantages of strong typing are lost at the native interfaces to C.
- Multiple levels of simulation (JVM, KVM, KVM on ISS, KVM on embedded processor) inevitably result in slightly different behavior of a single piece of code.

A verification technology that starts from interface definition and properties could have improved the testing. However, we are not aware of one that combines rigorous interface testing with multiple, heterogeneous programming languages. We also feel that such a technology must allow for fast design exploration.

For each of the individual ThumbPod components, a feasible bottom-up testing methodology was easy to find. But in the overall system however, the development paths of all these components overlap and the interfaces between those components are subject to constant change. The most effective strategy to improve integration was, not surprisingly, *team communication.* Figure 10 shows the traffic of project-related email as number of messages per project week. The release points are indicated, and show clear peaks. The focus shifts from academic interest to a working demonstrator. The fact that, in the end, Thumbpod works at all is due to a continuing alignment of the individual components to the overal system.

6. CONCLUSION

We discussed the design and testing process in the ThumbPod project. ThumbPod combines a stack of programmable machines, that were codesigned throughout the project. The key difficulty in testing was not in the individual components but in the overall system integration. Complex design should be driven out of the interfaces between components rather than out of the components themselves.

7. REFERENCES

[1] D. Hwang, P. Schaumont, Y. Fan, A. Hodjat, B.C. Lai, K. Sakiyama, S. Yang, I. Verbauwhede, *Design Flow for HW/SW Interface Acceleration Transparancy in the ThumbPod secure embedded system*, Proc. 40th DAC, Anaheim, CA, June 2003.

[2] "*The K Virtual Machine,"* SUN Microsystem, http://java.sun.com/products/cldc/ds/

[3] J. Yoshida, *"Electronic Passports Prep for Check-in,"* EETimes, June 9.

[4] V. Matyas, Z. Riha, "*Towards Reliable User Authentication through Biometrics", *IEEE Security and Privacy, May/June 2003.

[5] A. Sangiovanni-Vincentelli, G. Martin, *"Platform-based design and Software Design Methodology for Embedded Systems,"* IEEE Design and Test of Computers, November-December 2001.

[6] M. D. Garris, *"User's Guide to NIST Fingerprint Image Software (NFIS),"* NISTIR 6813, October 2001.

[7] http://www.ee.ucla.edu/~schaum/gezel

[8] S. Prabhakar, S. Pankanti, A.K. Jain, "*Biometric Recognition: Security and Privacy concerns*," IEEE Security and Privacy, March/April 2003.

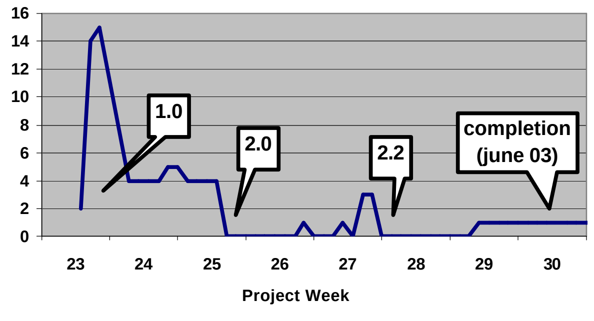


Figure 9: Bug Report Curve with release points

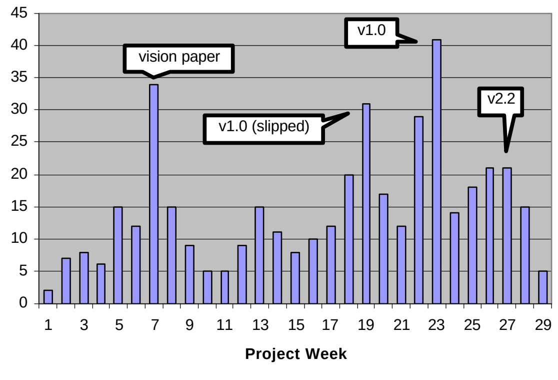


Figure 10: E-mail Traffic by Project Week