

A MULTI-TIERED SECURITY METHODOLOGY:
AN INVESTIGATION INTO LIGHTWEIGHT JAVA ACCESS
CONTROLS AND VERIFYING SECURE INFORMATION FLOW

by

Dustin Long

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE - COMPUTER SCIENCE

Dustin Long, Candidate

ADVISORY COMMITTEE

David Naumann, Advisor

Date

STEVENS INSTITUTE OF TECHNOLOGY

Castle Point on Hudson

Hoboken, NJ 07030

2006

A MULTI-TIERED SECURITY METHODOLOGY:
AN INVESTIGATION INTO LIGHTWEIGHT JAVA ACCESS CONTROLS
AND VERIFYING SECURE INFORMATION FLOW

We present a comprehensive security plan for a real-time software system, which intends to provide a full range of protection by covering multiple levels where potential weaknesses may lie. By using a proven, enterprise-level security framework at the broadest system view, down to using rigorous, theoretical techniques at narrower points, we intend to provide a deep and granular security methodology. For a test bed, we present the Codeblue system, a digital music project that uses remote hardware sensors to control its own playback. As it uses both wireless technology and a flexible plugin architecture, it is befit for discussion of security issues.

Author: Dustin Long

Advisor: David Naumann

Date: December 8, 2006

Department: Computer Science

Degree: Bachelor of Science

Acknowledgements

This research was financially supported by a grant from Telcordia Technologies and by the National Science Foundation program Research Experiences for Undergraduates (award CCF-0429894).

I'd like to acknowledge all the support and encouragement given by my professor, David Naumann. His consistent stream of advice was always helpful, and without his help I would not have had the opportunity to work on this project in the first place.

I also want to thank Susanne Wetzel for teaching me about Bluetooth and providing information about possible attacks, as well discussing design issues with Codeblue.

Much of my familiarity with Codeblue was acquired thanks to the previous maintainers, Oliver Gould and Scott Dryer. Also, Jared Cordasco, another ex-maintainer, provided help and guidance with the Codeblue hardware.

Help for working with Spec# was provided by Mike Barnett.

Contents

1	Background	5
2	Codeblue	7
3	Plugins and the Java Security Model	8
4	Problems With Java Security	9
4.1	Problem 1 - Granularity	9
4.2	Solution to Granularity - Dynamic Loading	9
4.3	Problem 2 - Performance	10
4.4	Solution to Performance - Proxy Objects	11
5	Foreign Code Formal Verification	14
6	Information Flow Analysis on Secure Information	15
7	Investigates Into Pair Composition	17
8	Alternative Verification Languages	22
9	Safety of Network Traffic	23
10	Discussion of Limitations	27
11	Feelings From the Author	28
A	Code for analysis experiments	29
A.1	Original C code	29
A.2	Translated to Java	31
A.3	Self-composed version	35

List of Figures

1	Part of Codeblue's dynamic class loader, with exceptions removed.	10
2	Plugin loading sequence in Codeblue.	13
3	Code sample exhibiting the need for security sensitive information flow	16
4	A pair-composed version of the m method.	18
5	Annotated version of the pair-composed m method.	19
6	Pair-composed method's signature using objects.	20
7	A method with security labels for parameters.	22
8	Calling a method using checks on parameters.	22
9	Spec# version of the pair-composed m method.	24
10	Sample C code to be checked.	25
11	Java code for Figure 10, annotated with JML.	26

1 Background

As long as software systems have existed, there have been associated security concerns over their safety and reliability. While there exist some easy to solve security flaws - such as revealing passwords in plaintext on screen, or executing code directly sent over a network - that can be handled using proper discipline, for the most part it is incredibly difficult to get total security right. Human code inspection, perhaps the most popular fix, can sometimes be effective, however insecure code often looks just as correct as secure code. Testing also has limited value, as insecure systems usually are exploited using hard to anticipate border cases. As Edsger Dijkstra once observed, “Testing can be used to show the presence of bugs, but never their absence!” When it comes to critical applications, we want a more reliable solution.

An alternative approach consists of formal code proof, to be exact, using automated tools along with program specifications to mechanically verify that a program does what is intended without any ill consequences. One such automated tool is the Extended Static Checker for Java (ESCJava [FLL⁺02]), which extends the Java language with JML annotations. Normally used to verify the correctness of Java code, such as pre/post-conditions, as well as null pointers avoidance, and maintaining array bounds, we instead use this tool to check security by phrasing a policy in terms of correctness assertions.

The technique we will use, based upon work by David Naumann[Nau06], transforms a piece of code into a self-composed pair. This entails copying the code into a second version, with renamed variable names, such that two subsequent runs of the original code is equivalent to one run of the pair-composed version. Then the data between the two parts of this composed pair is correlated using certain kinds of annotations. Data to remain secret

is allowed to vary between the two parts, which then yields our correctness condition: that non-secret data is never allowed to influence the state of secret data. By using ESCJava to prove this condition holds, we implicitly prove the safety of the original code, a property known as Non-Interference [SM03].

There are some efficiency concerns associated with code proof. At one extreme, the undecidability of the Halting Problem shows that we cannot verify all functions in the general case. But even moderately sized programs quickly become difficult to prove in a short time, as the proof techniques often involve theorem proving and model checking, both of which have large time/space requirements. Especially when real world code is verified, it becomes necessary to consider a large possible state space. This is the primary motivation for the work in this paper. Instead of using formal techniques to verify an entire software system from the bottom up, we divide it into levels, using faster but less stringent techniques towards the top, letting the hard cases fall through to the solver at the bottom. This makes the technique both powerful and reliable, and also requires minimal modification of the original source (with regards to adding annotations and creating specifications).

For the upper level of our system, we use a combination of the built-in Java access control mechanism with a Proxy design pattern for added efficiency. This is a novel usage of the Java security system that both gives the necessary security protections and also avoids being a performance burden on our real-time performance. The lower levels of our methodology use a formal verifier to prove that sensitive information leaks are prevented, using the Non-Interference property. This method is used directly on network code to check for safety, and also the possibility of using analysis on plugins was explored.

2 Codeblue

Codeblue is a combination of a hardware system and software system that was originally a computer engineering and computer science student project [HCV⁺03]. It has gone through many changes over the years, but its basic premise remains the same. A music synthesizer plays on a computer which has attached some wireless data receiver. Nearby, a dancer has attached certain sensors (light, pressure, acceleration) that are capable of sending data to the music playing machine. This wireless data then modifies the music, allowing the dancer to modify what they are dancing to using their own movements.

At first, the Codeblue system used only built-in functions to calculate how the dancer's movements would influence the music. However once more, and more varied, sensors were added to the setup, along with the possibility of multiple dancers, it was realized that a more open architecture was needed. Soon, the system was expanded to allow for arbitrary plugins. Each contained their own code that could modify the music without limits, by taking the input from the dancer's sensors, sent over wireless channels, and plugging it into arbitrary functions and calculations to decide what commands to send to the core music player. This is where the first set of security concerns became important.

Even though Codeblue is not the most security critical application, its architecture is similar enough to other projects which very well could be. It is an extensible system with a core system that wants to control access to protected resources. So finding way to secure and ensure the correctness of such a system goes to long way towards helping other similarly built systems.

3 Plugins and the Java Security Model

One of the greatest strengths of the Java Platform is its robust Security Model. It is built out of a combination of security concepts including Policies, Permissions, and stack inspections, providing enough flexibility to each application as it needs it. By default, all code can run whatever library functionality it pleases, which is clearly too lenient when plugins are involved. However, once a program instantiates a Security Manager, it becomes necessary to give Permissions to all high rights code. This is done using Policy files, each of which declares how client code is granted the ability to run Privileged operations. Examples of Permissions that can be granted include rights to files, rights to sockets, and other such built-in library calls. Also, Permissions can be custom defined by deriving new classes. In the case of Codeblue, we made a LowRights Permission class that gave access to common functionality (like adding notes), a HighRights Permission class (that can change the current song playing), and a MediumRights Permission class for things in between.

The standard mechanism for checking security in the Java Model is based around stack inspection. It works as so: when a piece of code performs some privileged action, the implementation of that action runs an inspection of the security rights of code running on the stack. Each piece of code within needs to have at least the requisite permissions for the given action. Thus, a low privilege piece of code cannot call code which eventually performs a high rights action.

For Codeblue, each plugin then would be given permissions based upon some set security policy. The Codeblue engine would then be the secure part of the codebase, and would perform the privileged operations itself, using the stack inspection capabilities to

check that a calling plugin has the requisite permissions. Thus the Codeblue core can provide a public API to the plugins, with controlled, privileged operations within itself, using the built-in Java framework as the basis of its upper security level.

4 Problems With Java Security

4.1 Problem 1 - Granularity

The first issue that surfaces with this plan is the lack of granularity involved with the specification of security policies for a given piece of code. As the Java Security Model was largely built for the web technology incarnation of Java, its characteristics are geared towards web development. For instance, policy files are able to specify permissions targeted at a single url (for the purposes of running remote code), or at a jar file (for the purposes of running downloaded packages), or for a single user. There is, however, no way to give permissions to an individual class file. As there's no guarantee that we'll be able to package each of Codeblue's plugins in individual jars, or hosted at different urls, it is clear that a better solution is needed.

4.2 Solution to Granularity - Dynamic Loading

The key to solving this problem comes from the Class Loader functionality of Java. The built-in class loading capabilities of the language are in fact surfaced for use by an application, allowing it to load and use new classes at runtime, assuming it has the applicable permissions. When such a class is loaded, it skips being given permissions from a policy file, ending up with no rights at all. Then the application which loads the class can itself build permissions and give them to the new dynamic class. This allows us to have complete granular control over the distribution of rights to newly loaded code.

```

protected Class createClassFromFile(String name, File f)
    throws ClassNotFoundException
{
    FileInputStream is;
    is = new FileInputStream(f);

    int size = (int)f.length();
    byte data[] = readClassData(is, size);

    ProtectionDomain protect = getClassProtection(f);

    return defineClass(name, data, 0,
                      data.length, protect);
}

```

Figure 1: Part of Codeblue's dynamic class loader, with exceptions removed.

Figure 1 shows a code sample from the dynamic loader used by Codeblue. It is a nice side effect that this code gives us run-time loading of new plugin classes, but its main motivation is to provide a way to granularly specify permissions for additions to the system. Since the Codeblue engine is considered to be trusted absolutely, it is given all permissions, allowing it to easily load new classes, determine the correct permissions from the policy files, and build permission objects.

4.3 Problem 2 - Performance

Another potential issue involved is the overhead involved with stack checking. Normally, the slowness caused by checking client code for permissions, while significant, is not prohibitive since most actions that are security sensitive are slow to begin with. For example, acquiring a system resource, opening a network socket, or modifying a file, are all time intensive enough that traversing the code stack is quick enough to be amortized away.

However, the security situation in Codeblue is different. Each plugin is supposed

to modify music while the system is playing it in real-time, while receiving data over a wireless network. The actions of a plugin may include things such as inserting new notes into a stream, testing the sound volume or equalizer levels, or even changing playlists. All such actions should require a certain level of permissions, as a malicious plugin could abuse them to disrupt the operation of Codeblue, for example by blasting the volume to dangerous heights, or erasing parts of the playlist when doing so if not desired. With this in mind, security checks will be happening during the entire usage of the system. However, even assuming a low quality or even synthesizer based audio stream, a plugin will need to run a few hundred times a minute, and will often need to perform calls into the Codeblue core. The more dangerous actions, like playlist modification, will occur rarely, but simpler actions, such as inserting notes and tweaking sounds could happen many times a second. At these rates, running a stack trace for each event quickly becomes prohibitive.

An additional observation is that, given the assumption of thin plugins, which do very little other than analyze data from a music stream, and then quickly act on it, it is clear that any given plugin will not change state much over time. In fact, the calling sequence from the music generator, to the client plugin, then back to the Codeblue engine for security sensitive actions, is rarely going to change whatsoever. Thus, any stack trace that may be performed before the first will be entirely redundant. Checking the same code path over and over is not going to make the system any more secure. This insight is the key to finding a solution for performance.

4.4 Solution to Performance - Proxy Objects

The solution used by Codeblue is to create a family of trusted Proxy Objects. Each such object is allowed access to Codeblue's API, and then surfaces an API of its own, with

a consistent interface across the entire family of proxies. The difference between these Proxy Objects is in how much of the Codeblue API they are given access to. For example, a Very Low Rights Proxy may only be able to raise and lower volume by a few notches, whereas a Full Rights Proxy can do anything from stopping the system to modifying the full range of sounds it produces. Once such a Proxy Object is instantiated, it is assumed to be stable and secure for the duration of the program run.

With this in place, the only singular security related operation a plugin needs to do is ask the Codeblue engine for a Proxy Object it can use to access the system. By requiring the client to call a request method explicitly, we can be certain that its code will be on the stack when the Codeblue engine decides to allocate a Proxy Object, thereby allowing the stack trace to take place. Therein the security rights of the plugin may be checked, and an appropriate accessor may be provided.

An important consideration here is that new security vulnerabilities need to not be introduced into the system by this methodology. An example of such a possibility is if the system allows a plugin to always ask for a new Proxy Object. If this were the case, a malicious piece of code could repeatedly ask for more proxies, with the potential result of a denial of service attack against the application. In this case, Codeblue avoids this pitfall by only allowing a plugin to request a Proxy Object during its construction.

Putting this all together, the entire operation is shown in figure 2. A need for a new plugin arises, and the system is asked to load the class dynamically. The Class Loader finds the class file, loads it, creates and applies permissions, and then gets ready to provide an interface. First, it begins to allow Proxies to be created. Then, it calls the plugin, passing a Proxy Factory that allows new Proxies to be created. The plugin needs to call back to the Factory requesting a Proxy for itself (it must do this now; it is its only chance).

The Class Loader then has the plugin on the stack, so it runs the stack privilege check, and allocates the corresponding Proxy Object. This is then returned to the plugin, which finishes its initialization, returning to the Class Loader. The Class Loader stops providing any Proxy Objects, then registers the new plugin, and finally returns to the main program for continued execution.

5 Foreign Code Formal Verification

One aspect of this system which can be improved for the sake of usability involves the classification of foreign plugins. For the sake of simple development, we could just implicitly give no rights to a new plugin, and then require a user to manually allow permissions. However, this is both error-prone and requires a user either have more knowledge or inspect code themselves, which can be difficult to do. Even though security and ease of use are often at odds, it is paramount that they not drift too far apart. From a user's perspective, the harder it is to employ secure techniques, the more often they will be ignored, ruining their entire purpose.

Another venue to be explored is to use formal verification techniques to observe an unknown plugin, and make some informed decision based upon how it appears. The intended purpose here is to automatically determine the rights that a chunk of code deserves based upon the safety of its actions. Thus explicit trust will become a direct function of how trustworthy the code is, providing a pleasant symmetry of terms.

This is one place where the multi-tiered approach comes in handy. We already know, by employing the Java Security Framework, that a given plugin will not be allowed to perform a large class of actions. Being able to open files, or create sockets, or load new classes, all require explicit permissions to be granted. Since our plugin verifier only needs

to worry about what Proxy Object to the Codeblue engine needs to be allocated, we needn't be concerned with such high-level attacks. Instead we can focus on much smaller, yet still dangerous, levels of action.

For one thing, we need to make sure that any passwords or other sensitive data a plugin collects is not accidentally leaked, and also that it does not try to covertly hide data in other forms to be collected at a later date, as is sometimes done with spyware and adware. This is where the Non-Interference Property comes into play.

6 Information Flow Analysis on Secure Information

Here we give an in depth explanation of the property of Non-Interference, which we use to prove the security of parts of the Codeblue system. The basic concept of this method relies upon being able to classify some data as low security, and some as high security. High security would apply to things like passwords, key, personal data, and other such secrets. Low security would be everything else. For any given section of code, it is allowable for low security data to be inputted, displayed, and influence other pieces of data. However, high security data should never influence the value of low data, since this implies that the high data is leaking to a publicly known location. For any two given runs of a program in which the low data starts off the same, but the high data may be different, the low data must end up the same in the end. An example of this technique in practice is illustrated in figure 3.

With this sample of code, “secret” is high security data, and everything else is low. A naive approach to enforcing this security constraint may be to make such high and low labels into explicit type declarations. For example, “x” and “y” could have the type “int_low” while “secret” has “int_high” with the following constraints:

```

int m(int x, int y, int secret)
{
    x = secret;
    if (secret % 2 == 1)
    {
        y = x * secret;
    }
    else
    {
        y = secret * secret;
    }
    int result = y - secret * x;
    return result;
}

```

Figure 3: Code sample exhibiting the need for security sensitive information flow

```

L : int_low, H : int_high
H := L is well-typed
L := H is not well-typed

```

This gets us most of what we want, but is too restrictive. As the code sample in figure 3 shows, “x” can temporarily get the value of some high level data as long as this does not influence the final result of the function. The pure type checking approach is too restrictive in this case.

What we want is just that the initial values of high level data cannot influence any low level data. The idea of labeling data as high or low is certainly on the right track, but using a type checker is not part of the full solution. Instead, we can specify this non-influence of data in other terms: if we were to run the program twice, with the same set of data for low level data for input, then no matter what values the high level data contains, the output of the two function runs should have no differences between the low level data. Stated symbolically for this particular function, we want to have

`m(x, y, secret) == m(x$, y$, secret$)` with `x==x$` and `y==y$`

(but not necessarily that `secret==secret$`). What we end up with is that the implicit labeling of high and low data is transformed into annotations that spell out our security requirement.

In order to fulfill this requirement of two runs of the same program, we take the original function, and duplicate it, with renamed variables in the second version. The goal is to make two separate yet identical runs turn into one single block of code, yet also guarantee that the runs do not individually affect each other. By renaming the variables we can be certain that the data sets will not interfere. By applying this transformation to the `m` method, we end up with the code in figure 4.

Then we use ESCJava annotations to ensure the security of the original code, as in figure 5.

When ESCJava is run on this sample and declares that the pre-condition leads to the post-condition, it implies that - from the perspective of our original security policy of “secret” being high level data - the code is safe. The only required modifications to the original code sample is a labeling of which data is low leveled, the rest is possible using a mechanical (in principle) translation.

7 Investigates Into Pair Composition

The work of David Naumann[Nau06] was to take this technique, and extend it to work with non-primitive types. This works by giving each object extra fields, “dash” to represent whether it belonged to the first part of the composition or the second, and also “mate” to point to its corresponding object. Then the mating relation was created to replace equality, in order to assure that each object correctly corresponds to its paired version. Using our

```
int Pair_m(int x, int y, int secret,
           int x$, int y$, int secret$)
{
    /* original */
    x = secret;
    if (secret % 2 == 1)
    {
        y = x * secret;
    }
    else
    {
        y = secret * secret;
    }
    int result = y - secret * x;
    /* dashed */
    x$ = secret$;
    if (secret$ % 2 == 1)
    {
        y$ = x$ * secret$;
    }
    else
    {
        y$ = secret$ * secret$;
    }
    int result$ = y$ - secret$ * x$;
}
```

Figure 4: A pair-composed version of the m method.

```

/*@ requires x == x$ && y == y$;
/*@ ensures result == result$;
int Pair_m(int x, int y, int secret,
           int x$, int y$, int secret$)
{
    /* original */
    x = secret;
    if (secret % 2 == 1)
    {
        y = x * secret;
    }
    else
    {
        y = secret * secret;
    }
    int result = y - secret * x;
    /* dashed */
    x$ = secret$;
    if (secret$ % 2 == 1)
    {
        y$ = x$ * secret$;
    }
    else
    {
        y$ = secret$ * secret$;
    }
    int resultP = y$ - secret$ * x$;
}

```

Figure 5: Annotated version of the pair-composed m method.

```

/*@ requires x.dash == false && x.mate == x$ &&
           x$.dash == true  && x == x$.mate &&
           y.dash == false && y.mate == y$ &&
           y$.dash == true  && y == y$.mate;
/*@ ensures result.dash == false &&
           result.mate == result$ &&
           result$.dash == true && result == result$.mate;
Integer Pair_m(Integer x, Integer y, Integer secret,
               Integer x$, Integer y$, Integer secret$)
...

```

Figure 6: Pair-composed method's signature using objects.

original function with “Integer” types instead of “int” types, as in figure 6.

Though the application is fairly trivial with regards to this function, it quickly becomes much more useful when we consider cases where objects are allocated based upon some initial program state, for example in a loop in order to fill an array of variadic size. It is in such cases that Pair Composition becomes powerful enough to find hard to detect information leaks.

A few optimizations were used in order to help with using this technique. The main one was merging parts of the pair back together whenever it could be proven that doing so would provide an isomorphic transformation. That is, although some parts of the composition would change and the pairs would partially overlap, an exact correlation could be drawn between program states and state transitions, to show that the final results were not affected and that data independence was preserved.

Such a manipulation very useful for conditional branches, since it reduced the complexity of the search space the theorem prover would have to cover. Whenever the condition to be tested relied only upon low security data, it would be safe to perform a merge,

letting us turn

```
A; if B then C fi; D; A'; if B' then C' fi; D'
```

into

```
A; A'; if B then C; C'; fi; D; D'
```

whenever B is only low security information. The reason this is safe to do is that, being low leveled, $B == B'$. Intuitively, the meaning behind this transformation is that low level data can safely effect control flow without concern for the different branches of execution, since the different code paths themselves become low security, and therefore cannot be influenced by the value of high security data.

Further research done on Pair Composition, in order to extend it to help with the Codeblue project, was the application on method calls. As verifiers like ESCJava work by analyzing a single method at a time, they ignore all details internal to those methods besides the one currently being analyzed. Instead, they only consider the pre/post condition of other methods, assuming they are correct whenever necessary. This is helpful for our needs, as it lets us phrase security information with regards to calling method using techniques in line with Pair Composition.

The stated purpose of our verifications is to ensure that high security data does not leak to low security places. So for method calls, we can treat parameter passing as similar to assignment to the parameters, the method body like any opaque - and already verified - block of code, and return values as assignments from the parameters. Our solution to ensuring the safety of a method was to add high/low security labels to each method call's parameters, and before a method call, ensure that all passed in values had the same or lower security types. Hence a function that manipulated password data like this:

Could have either paired or non-paired data passed for the first parameter, and only paired data passed for the second. It would be called as so:

```
String addSalt(String /*high*/ password, int /*low*/ salt)
...
```

Figure 7: A method with security labels for parameters.

```
String pwd = ...
int salt = ...
...
String ret = addSalt(pwd, salt);
...
String pwd$ = ...
int salt$ = ...
// do not assert that pwd$ is mated
//@ assert salt == salt$;
    addSalt(pwd$, salt$);
...
```

Figure 8: Calling a method using checks on parameters.

As the security precondition for the second call to `addSalt` references the state of “salt”, its value may not be changed in the interim between the two methods. This can be solved by either intertwining some parts of the original function call, or by creating new variables in the original undashed copy, and giving them copies of the values in “pwd” and “salt”.

8 Alternative Verification Languages

ESCJava is not the only annotation language used for this project. Research was also done into the Spec#[BLS05] language, an annotation and specification tool for C#. This was done mostly for investigations on how universalizable some of these methods could be to other platforms, and also how plausible it would be to have future portions of the project written in other languages.

Spec# mostly works the same as JML and ESCJava, aside from superficial syntax

differences, although there are also a few key differences under the surface. Spec# was mostly designed for verifying object contracts, specifically preconditions, postconditions, and invariants. There are a large number of primitives for handling ownership, reentrancy, and state modification. A Spec# version of the pair-composed `m` method is shown in figure 9. Note the need for the “expose” construct, which allows modification to an owned object.

9 Safety of Network Traffic

The last layer of safety considered is that of the wireless transmission. Codeblue uses Bluetooth to transmit data between sensors and receivers, specifically, what the reading is on each sensor (such as the intensity for a light sensor). We would like to have some security architecture in place to confirm that the usage of these devices does not give up too much information about the sensors that are sending, and additionally, that it will not compromise the Codeblue system. This is especially important as many vulnerabilities have been discovered in Bluetooth already.

This problem is not as profound as it may be in other applications, mainly as a consequence of the upper layers of security that we are already employing. Since it is known by this point that Codeblue does not trust plugins by default, we need not be concerned that bad wireless data could be snuck in to exploit part of the system. Codeblue is stable enough to withstand such attempts thanks to our multi-layered approach, however the loss of the wireless network would still be disruptive to system usage. So we would prefer that we could verify as much as possible with regards to this portion as well.

For our situation we needed a Java implementation that implemented enough of the Bluetooth stack to verify something interesting security-wise, such as PIN usage. Unfortunately, no open source Java codebase was available, so instead we opted for a C language

```

void Pair_m(int x, int y, int secret,
            int Dx, int Dy, int Dsecret)
  requires x==Dx && y==Dy;
{
  MInteger m_result, Dm_result;

  x = secret;
  if (secret % 2 == 1)
    y = x * secret;
  else
    y = secret * secret;

  m_result = new MInteger();
  expose (m_result) {
    m_result.val = y - secret * x;
  }

  Dx = Dsecret;
  if (Dsecret % 2 == 1)
    Dy = Dx * Dsecret;
  else
    Dy = Dsecret * Dsecret;

  Dm_result = new MInteger();
  expose (Dm_result) {
    Dm_result.val = Dy - Dsecret * Dx;
  }

  assert m_result.val == Dm_result.val;
}

```

Figure 9: Spec# version of the pair-composed m method.

```

err = read_link_key(sba, dba, key);
if (!err) {
    ba2str(dba, da);
    error("PIN code request for already paired device %s",
        da);
    goto reject;
}

```

Figure 10: Sample C code to be checked.

implementation, with parts translated into Java as closely as possible. Then annotations were added to this version as in figures 10 and 11, to both ensure no dangerous parts of code were in place, as well as to discover what would happen if some unsafe code did exist. This analysis of the Bluetooth stack was the most rigorous test for our information flow technique, and was the main application of this tool on the project.

Using this series of transformations, it was shown that all parts of the Bluetooth stack we investigated exhibited the Non-Interference Property with respect to our security policy. Since the paired method was successfully analyzed to show that paired data at the beginning was also paired at the end, it meant that our low level data was not influenced by high level data, and therefore high level data was not leaking. As a result, this code could be considered safe.

As a further investigation, we decided to test how the technique would handle an actual vulnerable piece of code. By taking the above sample and changing the last invocation of “error” from

```
error("PIN code request for already paired device ", da$);
```

to

```
error("PIN code request for already paired device ", key$);
```

```

        //@ set dba.mate = $dba;
        //@ set dba$.mate = dba;
        //@ set dba.dash = false;
        //@ set dba$.dash = true;
/* original */
    err = read_link_key(sba, dba, key); // modifies array
    // key is secret
    if (err != 0) {
        da = ba2str(dba);
        error("PIN code request for already paired device ", da);
        return;
    }
/* dash copy */
    err$ = read_link_key(sba$, dba$, key$); // modifies array
    // key$ is secret
    if (err$ != 0) {
        /*@ assert ((dba.mate == dba$) &&
                   (dba$.mate == dba) &&
                   (dba.dash == false) &&
                   (dba$.dash == true));
        */
        da$ = ba2str(dba$);
        //@ set da.mate = da$;
        //@ set da$.mate = da;
        //@ set da.dash = false;
        //@ set da$.dash = true;
        // check method security policy - precondition
        /*@ assert da.mate == da$ &&
                   da$.mate == da &&
                   da.dash == false &&
                   da$.dash == true;
        */
        error("PIN code request for already paired device ",
              da$);
        return;
    }
}

```

Figure 11: Java code for Figure 10, annotated with JML.

with the preceding annotations modified accordingly, the verifier caught this as an assertion violation, implying that the code had a vulnerability. This gives confidence that, in the case that there was unsafe code, this kind of analysis would catch it.

10 Discussion of Limitations

Overall, the entirety of this security architecture forms a robust and capable tool. The enforcement at higher levels helped to narrow the focus of analysis at lower levels, increasingly the practicality of what may be difficult to apply theoretical techniques. By also working on different types of insecurities we are covering more areas, avoiding redundancies, and forming complementary parts that mesh well into a cohesive whole.

As with any methodology, there are some limitations. One is purely technical; we use ESCJava for code proofs, but do so without knowing for sure either the ESCJava verifier is itself provably correct. It is, after all, a very complicated tool, and complex software nearly always has some bugs. Yet we need to make the assumption that its analysis is reliable for the sake of our work.

We also have to make the assumption that the Java platforms and security architecture are working as specified. This is a somewhat easier idea to allow, despite Java being vastly more complex, due to the fact that any failures in Java are much spectacular. Whereas a flaw in the verifier will give a false positive or negative, a bug in the platform would be much easier to spot. As such, it should do little to shake our confidence in our security methodology as a whole.

For the network security part, there was also the fact that we had to work with translated code instead of directly with real code from the wild. While this may have been a problem with trickier code, what was used for the pin code and key trading was very

straight forward code. The biggest change was turning C's stack allocated structures into Java objects, and direct memory copies into reference assignments. With only trivial, nested, object lifetimes, the transition from value semantics to reference semantics was fairly simple. It is still possible that error crept into the transformation, the likelihood is much smaller than it potentially could have been.

Another issue is social in nature. Any usage of security always entails some difficulty with usability, and while our project is no exception, the situation is not too bad. The first tough spot is that the programmer must manually specify which piece of data have what security level. If a mistake is made here, there's no guarantee that the verification techniques will be reliable. Similarly, there is room for error with policy files. In the case of plugins, a policy might need to be tweaked in order to fine-tune allowed actions. This has the potential that someone could give complete rights to a piece of code that doesn't deserve it. Contained within this is the possibility of social engineering type actions.

11 Feelings From the Author

On a personal note, working on this project was a welcome experience. It allowed me to get a wide, overarching view of security as a whole and work with it in the best way possible: at all levels at once.

A few unfortunate obstacles showed up during my work. For one, the verifier is still a work in progress, and while using it is often smooth and easy, there's not very much documentation or diagnostics for when things go wrong. Discovering why an analysis gives an unexpected result could at times be as difficult as debugging a medium scale program. I found a good technique was to insert spurious assertions into a piece of code (assert 1 == 2) and see if it triggered a failure; if it didn't then the problem was usually

earlier in the code. Doing this had much in common with the simplest debugging method of all: inserting extraneous print statements into source code.

Another setback involved the discovery of a suitable Bluetooth implementation. We only ended up using a Java to C translation because no suitable Java package could be found. Some amount of time was spent searching various Java libraries to see if any had code worthy of our project's aims, though this wasn't completely fruitless as it helped me become more familiar with the Bluetooth world. The C library that was eventually picked was the standard Linux package for Bluetooth, BlueZ. It was picked for its open-source status, its widespread usage, and the general clarity of the code contained within.

The theoretical work in particular was very engrossing. Being able to apply cutting edge techniques to real world problems is always eye-opening. This was definitely my favorite portion of the work, and the point from which I got the most out of.

As for future work on this area, I would be interested in seeing more parts of this project automated and integrated. Though the separate levels of security do exist, they all have to be employed manually, a single piece at a time. Having some tool that could perform the boilerplate work needed to connect everything together would make the entire solution more viable and easier to use as well.

A Code for analysis experiments

A.1 Original C code

This selection of code from the BlueZ stack uses security sensitive information, in particular the user's pin code and also the key used to encrypt network traffic. The logic is complex enough to be interesting yet simple enough to successfully analyze.

```
static void pin_code_request(int dev, bdaddr_t *sba, bdaddr_t *dba)
{
```

```

pin_code_reply_cp pr;
struct hci_conn_info_req *cr;
struct hci_conn_info *ci;
unsigned char key[16];
char sba[18], da[18], pin[17];
int err, pinlen;

memset(&pr, 0, sizeof(pr));
bacpy(&pr.bdaddr, dba);

ba2str(sba, sa); ba2str(dba, da);
info("pin_code_request (sba=%s, dba=%s)", sa, da);

cr = malloc(sizeof(*cr) + sizeof(*ci));
if (!cr)
    return;

bacpy(&cr->bdaddr, dba);
cr->type = ACL_LINK;
if (ioctl(dev, HCIGETCONNINFO, (unsigned long) cr) < 0) {
    error("Can't get conn info: %s (%d)", strerror(errno), errno);
    goto reject;
}
ci = cr->conn_info;

memset(pin, 0, sizeof(pin));
pinlen = read_pin_code(sba, dba, pin);

if (pairing == HCID_PAIRING_ONCE) {
    err = read_link_key(sba, dba, key);
    if (!err) {
        ba2str(dba, da);
        error("PIN code request for already paired device %s", da);
        goto reject;
    }
} else if (pairing == HCID_PAIRING_NONE)
    goto reject;

if (hcid.security == HCID_SEC_AUTO) {
    if (!ci->out) {
        /* Incoming connection */
        set_pin_length(sba, hcid.pin_len);
        memcpy(pr.pin_code, hcid.pin_code, hcid.pin_len);
        pr.pin_len = hcid.pin_len;
        hci_send_cmd(dev, OGF_LINK_CTL, OCF_PIN_CODE_REPLY,
                    PIN_CODE_REPLY_CP_SIZE, &pr);
    } else {
        /* Outgoing connection */
        if (pinlen > 0) {
            set_pin_length(sba, pinlen);

```

```

        memcpy(pr.pin_code, pin, pinlen);
        pr.pin_len = pinlen;
        hci_send_cmd(dev, OGF_LINK_CTL, OCF_PIN_CODE_REPLY,
                    PIN_CODE_REPLY_CP_SIZE, &pr);
    } else {
        /* Let PIN helper handle that */
        hcid_dbus_request_pin(dev, sba, ci);
    }
}
} else {
    /* Let PIN helper handle that */
    hcid_dbus_request_pin(dev, sba, ci);
}

free(cr);

return;

reject:
    free(cr);

    hci_send_cmd(dev, OGF_LINK_CTL, OCF_PIN_CODE_NEG_REPLY, 6, dba);

    return;
}

```

A.2 Translated to Java

```

class bdaddr_t
{
}

class pin_code_reply_cp
{
    bdaddr_t bdaddr;
    char [] pin_code;
    int pin_len;
}

class hci_conn_info_req
{
    bdaddr_t bdaddr;
    int type;
    hci_conn_info conn_info;
}

class hci_conn_info
{
    hci_conn_info out;
}

```

```

}

class hcid_opts
{
    char [] host_name;
    int auto_init;
    int security;
    int pairing;
    char [] config_file;
    char [] pin_code;
    int pin_len;
    int sock;
}

class PinException extends RuntimeException
{
    public PinException(String desc)
    {
    }
}

class charArrayHolder
{
    char [] charArray;
}

public class PinCodeRequestTester
{
    static final int ACL_LINK = 1;
    static final int HCID_PAIRING_ONCE = 2;
    static final int HCID_PAIRING_NONE = 3;
    static final int HCID_PAIRING_MULTI = 4;
    static final int HCID_SEC_AUTO = 5;
    static final int OGF_LINK_CTL = 6;
    static final int OCF_PIN_CODE_REPLY = 7;
    static final int PIN_CODE_REPLY_CP_SIZE = 8;
    static final int HCIGETCONNINFO = 9;

    static int pairing = HCID_PAIRING_MULTI;

    static hcid_opts hcid;

    public void pin_code_request(int dev, bdaddr_t sba, bdaddr_t dba)
    {
        pin_code_reply_cp pr = new pin_code_reply_cp();
        hci_conn_info_req cr = new hci_conn_info_req();
        hci_conn_info ci = new hci_conn_info();
        char [] key = new char[16];
        char [] sa = new char[18];
        char [] da = new char[18];
    }
}

```

```

char [] pin = new char[17];
int err; int pinlen;

ZeroMemory(pr);
pr.bdaddr = dba;

sa = ba2str(sba); da = ba2str(dba);
info("pin_code_request (sba=" + sa + ", dba=" + da + ")");

cr.bdaddr = dba;
cr.type = ACL_LINK;
if (ioctl(dev, HCIGETCONNINFO, cr) < 0) {
    error("Can't get conn info");
    return;
}
ci = cr.conn_info;

ZeroMemory(pin);
charArrayHolder pinHolder = new charArrayHolder();
pinlen = read_pin_code(sba, dba, pinHolder);
pin = pinHolder.charArray;

if (pairing == HCID_PAIRING_ONCE) {
    charArrayHolder keyHolder = new charArrayHolder();
    err = read_link_key(sba, dba, keyHolder);
    key = keyHolder.charArray;
    if (err != 0) {
        da = ba2str(dba);
        error("PIN code request for already paired device "
            + da);
        return;
    }
} else if (pairing == HCID_PAIRING_NONE)
    return;

if (hcid.security == HCID_SEC_AUTO) {
    if (ci.out != null) {
        /* Incoming connection */
        set_pin_length(sba, hcid.pin_len);
        memcpy(pr.pin_code, hcid.pin_code, hcid.pin_len);
        pr.pin_code = hcid.pin_code;
        pr.pin_len = hcid.pin_len;
        hci_send_cmd(dev, OGF_LINK_CTL, OCF_PIN_CODE_REPLY,
            PIN_CODE_REPLY_CP_SIZE, pr);
    } else {
        /* Outgoing connection */
        if (pinlen > 0) {
            set_pin_length(sba, pinlen);
            pr.pin_code = pin;
            pr.pin_len = pinlen;
        }
    }
}

```

```

        hci_send_cmd(dev, OGF_LINK_CTL, OCF_PIN_CODE_REPLY,
                    PIN_CODE_REPLY_CP_SIZE, pr);
    } else {
        /* Let PIN helper handle that */
        hcid_dbus_request_pin(dev, sba, ci);
    }
}
} else {
    /* Let PIN helper handle that */
    hcid_dbus_request_pin(dev, sba, ci);
}
}

public void ZeroMemory(pin_code_reply_cp memory)
{
    // STUB
}

public void ZeroMemory(char [] memory)
{
    // STUB
}

public char [] ba2str(bdaddr_t bd)
{
    // STUB
    return null;
}

public void info(String str)
{
    // STUB
}

public int read_pin_code(bdaddr_t sour, bdaddr_t dest,
                        charArrayHolder cah)
{
    // STUB
    return 0;
}

public void hcid_dbus_request_pin(int dev, bdaddr_t sour,
                                hci_conn_info ci)
{
    // STUB
}

public void hci_send_cmd(int dev, int ogf, int ocf,
                        int size, Object param)
{

```

```

        // STUB
    }

    public void set_pin_length(bdaddr_t target, int pinlen)
    {
        // STUB
    }

    public int read_link_key(bdaddr_t sour, bdaddr_t dest,
                            charArrayHolder cah)
    {
        // STUB
        return 0;
    }

    public int ioctl(int dev, int cmd, hci_conn_info_req cr)
    {
        // STUB
        return 0;
    }

    public void error(String msg)
    {
        // STUB
        return;
    }
}

```

A.3 Self-composed version

```

class bdaddr_t
{
    //@ ghost public bdaddr_t mate;
    //@ ghost public boolean dash;
}

class pin_code_reply_cp
{
    bdaddr_t bdaddr;
    charArray pin_code;
    int pin_len;
    //@ ghost public pin_code_reply_cp mate;
    //@ ghost public boolean dash;
}

class hci_conn_info_req
{
    bdaddr_t bdaddr;
}

```

```

        int type;
        hci_conn_info conn_info;
        //@ ghost public hci_conn_info_req mate;
        //@ ghost public boolean dash;
    }

class hci_conn_info
{
    hci_conn_info out;
    //@ ghost public hci_conn_info mate;
    //@ ghost public boolean dash;
}

class hcid_opts
{
    charArray host_name;
    int auto_init;
    int security;
    int pairing;
    charArray config_file;
    charArray pin_code;
    int pin_len;
    int sock;
}

class charArray
{
    char [] array;
    //@ ghost public charArray mate;
    //@ ghost public boolean dash;

    //@ requires size > 0;
    public charArray(int size)
    {
        array = new char[size];
    }
}

public class PinCodeRequestTester
{
    static final int ACL_LINK = 1;
    static final int HCID_PAIRING_ONCE = 2;
    static final int HCID_PAIRING_NONE = 3;
    static final int HCID_PAIRING_MULTI = 4;
    static final int HCID_SEC_AUTO = 5;
    static final int OGF_LINK_CTL = 6;
    static final int OCF_PIN_CODE_REPLY = 7;
    static final int PIN_CODE_REPLY_CP_SIZE = 8;
    static final int HCIGETCONNINFO = 9;
}

```

```

static int pairing = HCID_PAIRING_MULTI;

static hcid_opts hcid;

static int pairing$ = HCID_PAIRING_MULTI;

static hcid_opts hcid$;

/*@ ghost static public boolean safe_call;

//policy:
// key, pin, pinlen are secret
// everything else is public

/*@ requires dev == dev$;
    requires sba.mate == sba$ &&
           sba$.mate == sba &&
           sba.dash == false &&
           sba$.dash == true;
    requires dba.mate == dba$ &&
           dba$.mate == dba &&
           dba.dash == false &&
           dba$.dash == true;
*/
public static void
pin_code_request_pair(int dev, bdaddr_t sba, bdaddr_t dba,
                      int dev$, bdaddr_t sba$, bdaddr_t dba$)
{
    /*@ assume hcid != null;
        /*@ assume hcid$ != null;

/*=====original=====*/

    pin_code_reply_cp pr = new pin_code_reply_cp();
    hci_conn_info_req cr = new hci_conn_info_req();
    hci_conn_info ci = new hci_conn_info();
    charArray key = new charArray(16);
    charArray sa = new charArray(18);
    charArray da = new charArray(18);
    charArray pin = new charArray(17);
    int err; int pinlen;

    charArray nullCharArray = new charArray(1);

    ZeroMemory(pr);
    pr.bdaddr = dba;

    sa = ba2str(sba); da = ba2str(dba);

/*=====dashed copy=====*/

```

```

pin_code_reply_cp pr$ = new pin_code_reply_cp();
// secret

hci_conn_info_req cr$ = new hci_conn_info_req();
//@ set cr.mate = cr$;
//@ set cr$.mate = cr;
//@ set cr.dash = false;
//@ set cr$.dash = true;

hci_conn_info ci$ = new hci_conn_info();
//@ set ci.mate = ci$;
//@ set ci$.mate = ci;
//@ set ci.dash = false;
//@ set ci$.dash = true;

charArray key$ = new charArray(16);
// key is secret

charArray sa$ = new charArray(18);
charArray da$ = new charArray(18);
charArray pin$ = new charArray(17);
// pin should be secret
//@ set sa.mate = sa$;
//@ set sa$.mate = sa;
//@ set sa.dash = false;
//@ set sa$.dash = true;
//@ set da.mate = da$;
//@ set da$.mate = da;
//@ set da.dash = false;
//@ set da$.dash = true;
// pin is secret

int err$; int pinlen$; // pinlen should be secret

charArray nullCharArray$ = new charArray(1);
//@ assume nullCharArray != nullCharArray$;
//@ set nullCharArray.mate = nullCharArray$;
//@ set nullCharArray$.mate = nullCharArray;
//@ set nullCharArray.dash = false;
//@ set nullCharArray$.dash = true;

ZeroMemory(pr$);

pr$.bdaddr = dba$;
// sanity check - works because dba and dba$
//                where paired as parameters
/*@ assert pr.bdaddr.mate == pr$.bdaddr &&
           pr$.bdaddr.mate == pr.bdaddr &&
           pr.bdaddr.dash == false &&

```

```

        pr$.bdaddr.dash == true;
        */

/*@ assert ((sba.mate == sba$) &&
            (sba$.mate == sba) &&
            (sba.dash == false) &&
            (sba$.dash == true));
    assert ((dba.mate == dba$) &&
            (dba$.mate == dba) &&
            (dba.dash == false) &&
            (dba$.dash == true));
        */
sa$ = ba2str(sba$); da$ = ba2str(dba$);
/*@ set sa.mate = sa$;
/*@ set sa$.mate = sa;
/*@ set sa.dash = false;
/*@ set sa$.dash = true;
/*@ set da.mate = da$;
/*@ set da$.mate = da;
/*@ set da.dash = false;
/*@ set da$.dash = true;

// sanity
/*@ assert sa.mate == sa$ &&
        sa$.mate == sa &&
        sa.dash == false &&
        sa$.dash == true;
        */
/*@ assert da.mate == da$ &&
        da$.mate == da &&
        da.dash == false &&
        da$.dash == true;
        */

/*=====original=====*/

    cr.bdaddr = dba;
    cr.type = ACL_LINK;
    if (ioctl(dev, HCIGETCONNINFO, cr) < 0) {
        error("Can't get conn info", nullCharArray);
        return;
    }

    ci = cr.conn_info;

    ZeroMemory(pin);
    pinlen = read_pin_code(sba, dba, pin); // modifies array
    // pin and pinlen are secret

```

```

/*=====dashed copy=====*/

    cr$.bdaddr = dba$;
    // sanity
    /*@ assert cr.bdaddr.mate == cr$.bdaddr &&
        cr$.bdaddr.mate == cr.bdaddr &&
        cr.bdaddr.dash == false &&
        cr$.bdaddr.dash == true;
    */

    cr$.type = ACL_LINK;
    // sanity
    /*@ assert cr.type == cr$.type;

    /*@ assert ((dev == dev$) &&
        (HCIGETCONNINFO == HCIGETCONNINFO) &&
        (cr.mate == cr$) &&
        (cr$.mate == cr) &&
        (cr.dash == false) &&
        (cr$.dash == true));
    */
    if (ioctl(dev$, HCIGETCONNINFO, cr$) < 0) {

        // check method security policy - precondition
        /*@ assert nullCharArray.mate == nullCharArray$ &&
            nullCharArray$.mate == nullCharArray &&
            nullCharArray.dash == false &&
            nullCharArray$.dash == true;
        */
        error("Can't get conn info", nullCharArray$);
        return;
    }

    /*@ assume cr.conn_info != cr$.conn_info;
    /*@ assert cr.conn_info != cr$.conn_info;

    /*@ set cr.conn_info.mate = cr$.conn_info;
        set cr$.conn_info.mate = cr.conn_info;
        set cr.conn_info.dash = false;
        set cr$.conn_info.dash = true;
    */

    ci$ = cr$.conn_info;
    /*@ assert ci.mate == ci$ &&
        ci$.mate == ci &&
        ci.dash == false &&
        ci$.dash == true;
    */

    ZeroMemory(pin$);

```

```

// secret

pinlen$ = read_pin_code(sba$, dba$, pin$); // modifies array
// pin$ and pinlen$ are secret

/*=====MERGE=====*/
// because [pairing] is public

    if (pairing == HCID_PAIRING_ONCE) {

/*=====original=====*/

        err = read_link_key(sba, dba, key); // modifies array
        // key is secret
        if (err != 0) {
            da = ba2str(dba);
            error("PIN code request for already paired device ",
                da);
            return;
        }

/*=====dash copy=====*/

        err$ = read_link_key(sba$, dba$, key$); // modifies array
        // key$ is secret

        if (err$ != 0) {

/*@ assert ((dba.mate == dba$) &&
            (dba$.mate == dba) &&
            (dba.dash == false) &&
            (dba$.dash == true));
        */
            da$ = ba2str(dba$);
            //@ set da.mate = da$;
            //@ set da$.mate = da;
            //@ set da.dash = false;
            //@ set da$.dash = true;

// check method security policy - precondition
/*@ assert da.mate == da$ &&
    da$.mate == da &&
    da.dash == false &&
    da$.dash == true;
*/
            error("PIN code request for already paired device ",
                da$);
            return;
        }
    }
}

```

```
        // trimmed for efficiency
    }

    public static void ZeroMemory(pin_code_reply_cp memory)
    {
        // STUB
    }

    public static void ZeroMemory(charArray memory)
    {
        // STUB
    }

    //@ ensures \result != null && \fresh(\result);
    public static charArray ba2str(bdaddr_t bd)
    {
        // STUB
        charArray result = new charArray(1);
        return result;
    }

    public static void info(String str)
    {
        // STUB
    }

    public static int read_pin_code(bdaddr_t sour, bdaddr_t dest,
                                    charArray ca)
    {
        // STUB
        return 0;
    }

    public static void hcid_dbus_request_pin(int dev, bdaddr_t sour,
                                             hci_conn_info ci)
    {
        // STUB
    }

    public static void hci_send_cmd(int dev, int ogf, int ocf,
                                    int size, Object param)
    {
        // STUB
    }

    public static void set_pin_length(bdaddr_t target, int pinlen)
    {
        // STUB
    }
}
```

```

public static int read_link_key(bdaddr_t sour, bdaddr_t dest,
                               charArray ca)
{
    // STUB
    return 0;
}

/*@ requires cr != null;
    @ ensures cr.conn_info != null;
public static int ioctl(int dev, int cmd, hci_conn_info_req cr)
{
    // STUB
    cr.conn_info = new hci_conn_info();
    return 0;
}

public static void error(String msg, charArray about)
{
    // STUB
    return;
}
}

```

References

- [BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, volume 3362 of *LNCS*, pages 49–69, 2005.
- [FLL⁺02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.
- [HCV⁺03] Dennis Hromin, Michael Chladil, Natalie Vanatta, David Naumann, Susanne Wetzels, Farooq Anjum, and Ravi Jain. CodeBlue: a Bluetooth interactive dance club system. In *IEEE Globecom*, 2003.

- [Nau06] David A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in LNCS, pages 279–296, 2006.
- [SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.