Improving and Extending Verigames

Stephanie Dietzel
Master's Thesis
June 2013

Abstract:

In this paper I present my work on Verigames, a system to turn a formal verification problem into a puzzle game. I extend a game that detects null pointer errors, create a testing framework, contribute a case study on a type system for trusted and untrusted values, extend that trusted type system to create 9 new type systems, and present a conversion of a new type system into simple game.

1 Introduction

Formal verification can guarantee correctness of code with regard to certain properties, or discover new bugs. However, formal verification requires the labor of skilled engineers which is often prohibitively expensive.

Verigames[1] is a system that presents verification as a visual puzzle game that anyone can play in order to lower the skill level needed to formally verify code. The system takes as input a program and a type system and translates it into a puzzle game. Then, the solved puzzle game configuration translates back into source code with inserted type annotations. The annotated code can then be verified using the Checker Framework[2], a pluggable type checker.

Verigames aims to create type systems and games to verify 25 important security properites, specified as some of the most common or serious security errors by the Common Weakness Enumeration (CWE)[3].

In game form, variables are represented as pipes that may be wide or narrow. Methods are boards with networks of pipes flowing through them. When methods are called, a subboard is created within the board of the calling method. Narrow pipes may flow into wide pipes, but wide pipes may not flow into narrow pipes. Players may manipulate the widths of pipes to eliminate these conflicts. A set of boards is generated from an input program, and the player adjusts the widths of the pipes in the network until there are no conflicts in the flow of the pipes. The widths of the pipes in the final configuration map back to properties of each variable in the program.

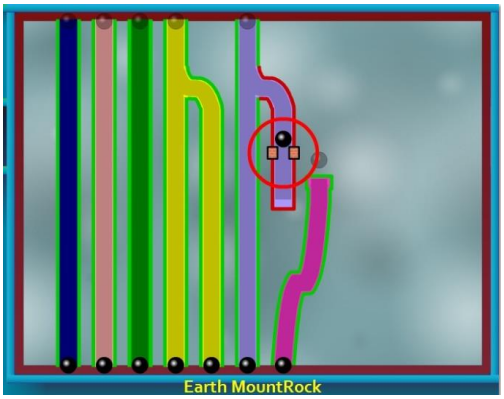


Figure 1. A board in Verigames

In this paper, I discuss my work on Verigames, including the @KeyFor relationship in the Nullness type system, a testing framework, a Trusted type system case study, 9 additional trusted type systems, and a conversion of the Lock type system into a simple game. Though my work has been on quite varied problems and aspects within Verigames, it has all been with a goal of helping Verigames progress towards a full-fledged system that uses game play to lower barriers to formal verification of important properties.

2 Map.get() and the @KeyFor Relationship in Verigames

The first type system to be turned into a game was the Nullness type system, which expresses the @Nullable :> @NonNull relationship. When a variable has the type annotation @NonNull it is guaranteed to never be null, and can be safely dereferenced. A @Nullable variable may or may not be null, and cannot be safely dereferenced or assigned to a @NonNull variable.

An example of the Nullness type system:

```
Object foo;
@NonNull Object bar;
foo.toString(); // ERROR!
bar.toString(); // OK ☺
```

In the nullness game, @Nullable variables have wide pipes and @NonNull variables have narrow pipes. This section discusses the @KeyFor feature of the Nullness type system and my work to implement the feature in the Verigames Nullness inference game.

2.1 Motivation for @KeyFor

In Java, there are two possible cases where Map.get() returns null. First, null is the actual value associated with the key. Second, the key is not stored in the map. This distinction is important, and means that in the Checker Framework's Nullness type system, the return type of Map.get() must always be @Nullable in order to be conservative, even for Maps with @NonNull values. This is frustrating when using Map.get() with a known key in the map; since the value will be no-null at runtime but the CheckerFramework still gives errors that the values is @Nullable.

To work around this, the Nullness type system uses an additional annotation to mark the known keys for a map. When Map.get() is called using a key that is annotated as @KeyFor that map, the return value type will be the same as the map's value type. For example:

```
Map<String, @NonNull String> myMap;
@KeyFor("myMap") String myKey;
@NonNull String myVal = myMap.get(myKey); // No error!
```

The @KeyFor annotation allows developers to indicate relationships between map variables and key variables, and avoid false positive errors while type-checking their code.

2.2 Representation in Verigames Game Play (PipeJam Classic)

The @KeyFor annotation is an important part of the Nullness type system, and needs to be included when converting the type system into a game for Verigames. To do this, we use colored stars to represent the relationship between different pipes (variables). Each pipe has a color, and to indicate a pipe is a key for a map pipe, the key pipe will be stamped with stars that are the same color as the map pipe. A pipe can be a key for more than one map, and can therefore have stars of more than one color.

A call to Map.get() is different from a normal method call in the Nullness type system, so it will be different from a normal subboard in Nullness Game. We have a special Get Node which takes four inputs: the map pipe, the map's key type pipe, the map's value type pipe, and the argument pipe. On the inside of the Get Node is a mechanism which checks the argument pipe for stars the same color as the map pipe. If the argument pipe has the right

color stars, it is a key for that map and the mechanism allows the map's value type pipe to flow through as the output pipe. If the argument pipe is missing the right stars, the mechanism creates a wide pipe as the output pipe. The player manipulates the stars on pipes by selecting a pipe and choosing from a color-wheel-like mechanic.

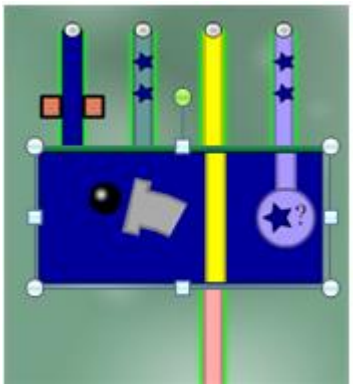


Figure 2. KeyFor

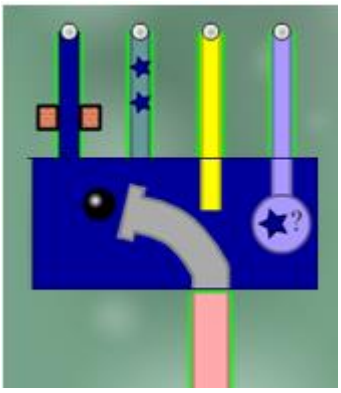


Figure 3. Not KeyFor

Above, in Figure 2 the argument (rightmost pipe) has a blue star indicating that it is a key for the blue map (leftmost pipe) and the value type flows through the output. In Figure 3 the argument does not have blue stars, so it is not a key for the blue map and a wide pipe flows through the output.

2.3 Limiting Stars to Sensible Options

Because the code we convert into games is unannotated, none of the Key-For relationships are explicitly indicated. We don't want the player to be able to add any color star to any pipe, because that could translate back into nonsensical relationships, such as a String as a key for a map with Integer keys. The player is doing a type inference task, so we want to restrict their star choices to the underlying types that make sense. Only stars that represent maps' pipe colors should be added, and the color options should be limited to the colors of pipes that match the key type of the map, which are all potential keys. When processing the source code, we must compute the variables that are possible keys for each map and record this 'Possible Key-For' relationship to be included in the game.

2.4 Implementation

In order to compute the 'Possible Key-For' relationships in a program, I added some extra logic to the Nullness type system's existing code. The Verigames Nullness type system uses the visitor pattern to traverse the abstract syntax tree of the input code. When it visits each block of code (each set of curly braces, roughly), I keep a set of variables and a set of maps that are in scope. I examine variables within the block, and then traverse up the code tree to find variables declared at an enclosing scope. Then, at the end of each block, for each map I check if any of the variables match the type of the map's key. If the variable type and key type are compatible, I record that the variable has a 'Possible Key-For' relationship with the map. Primitives are not shown by default in Verigames, but autoboxing means that an int could be a possible key for a map with a key type of Integer. I deal with this by adding

all variables in scope to the set of variables as we traverse up the tree. Then, when comparing to the maps' key types, I simply check for primitive variables and use their boxed types.

To be able to record that relationship, and later process it while building the game, I had to modify the Checker Inference Framework, which saves information about a program needed to turn it into a game. The Checker Inference Framework records information about a program by keeping a list of constraints. I created an additional Possible Key-For constraint, which simply includes a map variable and potential-key variable and indicates that they have a Potential-KeyFor relationship. I also added more information, such as receiver type arguments, to the constraint that records method invocations. This allows us to tell the difference between a normal method call and a call to Map.get when the constraints are turned into a game, and decide whether to use a subboard or a Get Node.

3 Testing Framework

Next, I created a basic testing framework for Verigames. While the Checker Framework and Checker Inference Framework had existing tests, testing for Verigames was completely ad-hoc. Developers could create small programs, run them through the system, and examine the output manually for correctness. This had worked early in the project, but as more people became involved and had different levels of understanding for different pieces of the system, another approach became necessary. I created a test framework and initial set of tests that checked two crucial points in the Verigames system, were easily extensible for additional type systems, and could be used by any developer on the project.

3.1 Types of Tests

I created two types of tests, constraint and XML, which cover two important points in the Verigames end-to-end system. Each set can be run independently, for either all the type systems or a specific one.

3.1.1 Constraint Tests

Constraint tests are used to verify that a Verigames type system records the correct constraint information in the Checker Inference Framework as it traverses the abstract syntax tree of the input program. Each test runs an inference, prints the generated constraints, and compares them to a file of expected constraints. Constraint tests use the first half of the source-to-game pipeline by checking how an input program is converted to a set of constraints.

3.1.2 XML Tests

XML tests verify that constraints are correctly converted into an XML representation of a game. Each test runs an inference, converts the constraints into a game, outputs the game to XML format and compares the generated XML to a file of expected XML. Because layout

information may change slightly from run to run, layout is removed from the XML files before the comparison. XML tests use the entire source-to-game pipeline, from input program to constraints to game.

3.2 Adding New Tests

With the test framework, it is simple to add additional tests and type systems. To add a new test, the developer simply places a program and an expected output file named 'expected_<typesystem>' into a directory of tests for each type of test. Expected output for multiple type systems can be stored in the same directory as the input program. To add a new type system, a developer must simply add an additional line to the test script.

3.3 Results

Within a few days of creation, the XML tests revealed undesirable behavior in the Verigames system. The same input program could generate different XML between runs, and this showed up as a flaky XML test. Though neither version of XML was an incorrect representation of the game, this challenged our assumption that the same program would produce the output at every step of the Verigames pipeline. I investigated and replaced several data structures with nondeterministic iterators with deterministic equivalents, and our system's behavior once again (or perhaps for the first time) matched our assumptions about deterministic behavior.

A set of constraint and XML tests for both the Nullness and Trusted type systems now runs nightly.

4 Trusted Type System Case Study – Apache Chukwa

Eventually, Verigames aims to create type systems and games that verify code with respect to 25 common security issues. Many of these involve mistakenly using an untrusted value where a trusted one is needed. Because of this, Verigames includes a Trusted type system which expresses an @Untrusted :> @Trusted relationship. The Trusted type system is a simple type system that distinguishes between objects that may or may not be trusted and objects that are definitely trusted. By default, objects are considered @Untrusted and must be explicitly annotated as @Trusted. A trusted object can be used anywhere an untrusted one can, but a variable with an @Untrusted type cannot be used if variable with an @Trusted type one is required. A program can require trusted values in certain situations to create a guarantee about some security property. The Trusted type system is intended to be used as a base for other more specific type systems, but first we needed a better understanding of how it applies to real code. Type-checking case studies give insight into the usability and expressiveness of a type system, help to understand how to improve it in the future, and give us an understanding of how the properties of a type system appear in large codebases in practice[4]. I did a type-checking case study of the Trusted type system on the Apache Chukwa, a system for monitoring distributed system that uses queries in

many places. These queries must use @Trusted values, and give a good starting place for annotating the codebase.

The Trusted type system is extremely simple to understand, but I found it was not simple to use, at least for this particular codebase. I found myself casting Untrusted Strings to Trusted Strings very frequently, suppressing warnings on common false positives, and writing what felt like extra annotations.

4.1 Reasons for Casting

4.1.1 Unannotated Libraries

Some library functions should always return a Trusted value. For example, executing a query creates a ResultsSet, which contains all the rows and columns from the database that match the query. Any information that comes from a ResultSet should be Trusted because it comes from the database, a Trusted source. However, without an annotated version of ResultSet, the information must be cast to @Trusted. This can be addressed with a stub file.

```
@Trusted String result = (@Trusted String) resultSet.getString();
```

4.1.2 Manipulating Trusted Strings

Often, new Trusted Strings are built from pieces of other Trusted Strings. In one case, the developers had templates for queries and used regular expressions to create real queries out of these templates, replacing placeholders with things like a table name. If the table name is Trusted, then the overall query would be Trusted as well and needed to be cast as such. Methods such as String.replace(), String.split(), etc. needed to have their return values cast to @Trusted on a case-by-case basis.

I also saw StringBuffers used to create Strings that were logically Trusted, such as to insert a partition number in the middle of a Trusted string. While I could verify these cases logically, the resulting string still needed to be cast. Adding polymorphism to the Trusted type system may be useful in general, but in this case it would not guarantee that all modifications to a particular StringBuffer used @Trusted Strings.

```
@Trusted String str = (@Trusted String) buffer.toString();
```

4.1.3 Primitives concatenated into Trusted Strings

In a few cases, queries were built up using concatenation and included a partition number. These numbers were a result of a calculation, and the Trusted type system does not understand numeric manipulation. So, it did not work to just annotate these numbers as Trusted at their declaration; I would still have to cast them at every mathematical operation. The easiest solution was to cast them to @Trusted once at the end as they were being used in the query.

```
@Trusted String query = "foo " + (@Trusted int) partition + " bar";
```

4.2 Other Common Issues

4.2.1 Trusted String is initially null

In several places, a certain String could be either null or a few specific String literals. If it was non-null, it was one of those Trusted String literals and used to create a new Trusted String. Or, a string was simply declared as null, and then later in the method assigned to a literal using a series of if-else statements. These initially-null Strings are clearly Trusted, but assigning a Trusted String as null is a type error. Since they would never be used when null, I suppressed the error.

```
@SuppressWarnings("trusted")
@Trusted String str = null;
…
if(someCondition)
      str = "foo";
else
      str = "bar";
…
@Trusted String query = "someQuery" +str;
```

4.2.2 Copying a Trusted String

A copy of a Trusted String must also be a Trusted String, but the checker does not understand this. It was slightly irritating to keep getting the following false positive errors, even though they were trivial to suppress. Here, adding polymorphism to the Trusted type system would fix the problem.

```
this.query_state = new String("read");
                           ^
  found   : @Untrusted String
  required: @Trusted String

public void setJobName(@Trusted String s) {
    this.jobname = new String(s);
                       ^
  found   : @Untrusted String
  required: @Trusted String
```

4.3 Did I find any errors?

I found a few slightly questionable things in the code, but nothing concrete.

In two places, a Trusted String is taken directly out of an HTTP Request object without verification. That might be an issue because an attacker could potentially build a request with anything in it, but neither place was ever used in the codebase. It might be dead code.

In another case, a list of queries is read in from a configuration file and then each is executed. In a README which explains the configuration files, it says this file should never be modified. The permissions on this particular file are "-rw-r--r--", so it is read-only for

everyone except the owner. I'm not sure if this is a security concern or not. The owner of the file could insert malicious queries, but then he would just be hacking himself.

4.4 What might this look like as a game?

Given the number of casts that I had to use, one possibility is that players need to use a lot of buzzsaws. Perhaps they will choose to put buzzsaws as close to the problem as possible. Or, alternatively, they might choose to make more pipes narrow than necessary, and stick buzzsaws at random. It might be very easy for players to get lost; this happened to me as an annotator. A small logic mistake made me think something needed to be Trusted when it didn't, and this led to a chain-reaction of variables that would need to be Trusted. Once I realized that I had an argument to the main method flowing into a Trusted variable, I knew something was wrong and backtracked to find my mistake. I expect players may experience a similar need to backtrack or rethink their higher level strategy, but I'm not sure whether the abstraction of the game would help or hinder them with this. I had the advantage of being able to read the code and reason about it in ways that revealed my mistakes. Perhaps if they are only presented a series of pipes, players would not have a similar intuition about why an Untrusted argument to main does not make sense and they would leave mistakes in their completed solution.

5 Additional Trusted Type Systems

After the Trusted case study, I moved on to creating type systems for the intended 25 security properties. I started with those that were most well-understood: the type systems based on the Trusted type system. I analyzed the 25 security properties and identified 9 that could be verified using a more specific version of the Trusted type system.

Each type system that is built using the Trusted type system describes a security property, and consists of two qualifiers, where the a variant of @Untrusted (the default) represents a variable which may or may not be secure, and a variant of @Trusted represents a variable that is definitely secure. Only String concatenation between two trusted Strings creates a trusted result; all other combinations create an untrusted result. Each system also has a jdk.astub file which describes how the JDK would be annotated with the type system qualifiers, and is used in type checking.

5.1 The Trusted Type Systems

5.1.1 Trusted

Qualifiers: @Untrusted :> @Trusted
Security Property: CWE-807 Reliance on Untrusted Inputs in a Security Decision

This type system is the original trusted type system, and may be used if one of the more specific type systems does not make sense for a given situation. By default, variables are

@Untrusted. To use a @Trusted variable, it must be included in the jdk.astub file or be explicitly annotated as @Trusted.

5.1.2 OsTrusted

Qualifiers: @OsUntrusted :> @OsTrusted
Security Property: CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

This type system describes @OsUntrusted variables, which may or may not be sanitized for use in OS commands, and @OsTrusted variables which have definitely been sanitized for use in OS commands. Code that creates OS commands should require @OsTrusted inputs. Some routines, such as `Runtime.exec`, `Runtime.load`, etc. have already been annotated in the jdk.astub file.

5.1.3 NotHardCoded

Qualifiers: @MaybeHardCoded :> @NotHardCoded
Security Property: CWE-798 Use of Hard-coded Credentials

This type system describes @MaybeHardCoded variables that may have been hard-coded into the source code, and @NotHardCoded variables that have been generated instead of hard-coded. An authentication routine can require that credentials are @NotHardCoded. Also, in this type system String concatenation returns an @Trusted result as long as at least one of the concatenated Strings is @NotHardCoded, since if one part has been generated then the concatenated result will vary as well.

5.1.4 Random

Qualifiers: @MaybeRandom :> @Random
Security Property: CWE-330: Use of Insufficiently Random Values

Some random number generators are not cryptographically secure. This type system describes @MaybeRandom variables that might be random and @Random variables that are definitely securely random. Routines can require secure @Random variables when necessary. The jdk.astub file specifies values returned by SecureRandom to be @Random.

5.1.5 Encrypted

Qualifiers: @Plaintext :> @Encrypted
Security Property: CWE-311 Missing Encryption of Sensitive Data,
CWE-327 Use of a Broken or Risky Cryptographic Algorithm

@Plaintext describes any variable that may or may not be encrypted. @Encrypted describes a variable that is known to be encrypted. The utilities provided by the JDK for encryption are modal, and may return either plaintext or an encrypted value. For this reason they are not included in the jdk.astub file, and annotating routines that provide @Encrypted values is left to the user for their specific use cases.

5.1.6 OneWayHashWithSalt

Qualifiers: @MaybeHash :> @OneWayHashWithSalt

Security Property: CWE-759 Use of a One-Way Hash without a Salt

This type system describes @MaybeHash variables that may or may not be a hash, and @OneWayHashWithSalt variables that are definitely hashes with a salt. Routines can require values that are @OneWayHashWithSalt to prevent against dictionary attacks. There are some hashing routines in the JDK, but these allow a salt to be null so annotating hashing routines is currently left to the user. In the future, a more sophisticated AnnotatedTypeFactory for this type system may help. To guarantee that a salt is random, this type system can be combined with the Random trusted type system.

5.1.7 SafeFileType

Qualifiers: @UnknownFileType :> @SafeFileType
Security Property: CWE-434 Unrestricted Upload of File with Dangerous Type

This type system describes @UnknownFileType variables that may or may not be a safe file type for upload and @SafeFileType variables that definitely are a safe file type for upload. Upload routines can require that uploaded values are verified to be @SafeFileType before they are used.

5.1.8 Internal

Qualifiers: @Internal :> @Public
Security Property: CWE-209: Information Exposure Through an Error Message

This type system describes @Internal variables that may or may not be appropriate to display to the end-user, and @Public variables that are definitely alright to display. Error handling code can require that the error messages it shows be annotated as @Public. In the jdk.astub file, PrintStreams are annotated to require @Public variables for printing.

5.1.9 Encoding

Qualifiers: @UnknownEncoding :> @AppropriateEncoding
Security Property: CWE-838: Inappropriate Encoding for Output Context

This type system describes @UnknownEncoding variables that have an unknown encoding, and @AppropriateEncoding variables that have the correct encoding needed to be safely used by another system component. Note that @AppropriateEncoding could be renamed to reflect the application-specific required encoding, for example UTF-8.

5.1.10 Download

Qualifiers: @ExternalResource :> @VerifiedResource
Security Property: CWE-494 Download of Code Without Integrity Check

This type system describes @ExternalResource variables that have been downloaded, such as external libraries, etc., and @VerifiedResource variables that have been downloaded and gone through a verification routine to guarantee they have not been tampered with. Then, when using external resources, the program can require @VerifiedResource variables.

5.2 Examples of Stub Files

Each type system has a corresponding JDK stub file, which contains "stub classes" with annotated method signatures but no method bodies. A type-checker uses the annotated signatures at compile time, and uses the annotations specified in the stub file instead of default annotations. This reduces the number of false positive errors, eases the task of code annotation for the developer, and provides a starting place for annotating code.

This stub file is for the ostrusted type system, and describes which methods in the JDK require @OSTrusted values:

```
import ostrusted.quals.*;

package java.lang;

class Runtime {
      public Process exec(@OsTrusted String command);
      public Process exec(@OsTrusted String[] cmdarray);
      public Process exec(@OsTrusted String[] cmdarray, String[] envp);
      public Process exec(@OsTrusted String[] cmdarray, String[] envp, File
dir);
      public Process exec(@OsTrusted String command, String[] envp);
      public Process exec(@OsTrusted String command, String[] envp, File
dir);

      public void load(@OsTrusted String filename);
      public void loadLibrary(@OsTrusted String libname);
}

class ProcessBuilder {
      public ProcessBuilder command(List<@OsTrusted String> command);
      public ProcessBuilder command(@OsTrusted String... command);
      public Map<String, @OsTrusted String> environment();
}

class System {
      public static String setProperty(String key, @OsTrusted String value);
      public static void load(@OsTrusted String filename);
      public static void loadLibrary(@OsTrusted String libname);
}
```

5.3 How are they implemented?

Each trusted type system is built on a supertype type system. The actual logic is written once, in the TrustedChecker, TrustedVisitor, TrustedAnnotatedTypeFactory, and TrustedGameSolver. Each additional trusted type system must simply define its qualifiers and create a Checker. For more specialized behavior, it could also create an AnnotatedTypeFactory.

The class TrustedChecker has two AnnotationMirror fields, representing TRUSTED and UNTRUSTED. Each Checker for a trusted type system extends the TrustedChecker, creates

AnnotationMirrors based on its qualifiers, and sets the TRUSTED and UNTRUSTED fields to use those new AnnotationMirrors.

When a trusted type system needs to have different behavior from the parent Trusted type system, I add a new AnnotatedTypeFactory for that type system which extends the TrustedAnnotatedTypeFactory, and then override any necessary functions to create the desired behavior.

5.4 Results

I created 9 of the 25 desired type systems to address security problems by extending the Trusted type system. Because these type systems use the Trusted parent classes, they are already ready to be turned into games using the Trusted Game Solver.

6 Lock Checker Game

6.1 Motivation

Verigames seeks to provide a correctness guarantee about 25 important security problems by creating a type system that expresses the property and turning it into a game. One such problem is CWE-362: Concurrent Excecution using Shared Resource with Improper Synchronization, also known as 'race condition'. Race conditions are often hard to reason about and debug, and can lead to improper behavior in real systems. The Checker Framework's Lock type system expresses exactly this property.

6.2 The Lock Type System

The Lock type system expresses the relationship between variables, and ensures that the appropriate lock is held every time a variable is accessed.

The Lock type system uses two annotations:

@GuardedBy is a type qualifier placed on the locked variable. It takes as an argument the name of the variable's corresponding lock. In the example below, `lock` must always be acquired before `foo` is accessed.

@Holding is a method annotation that indicates a lock must be held before the method is called. In the example below, `lock` must be acquired before calling `holdingMethod`.

```
Object lock;
@GuardedBy("lock") Object foo;

@Holding("lock")
public void holdingMethod(){
      foo.toString();   // OK, lock is guaranteed to be held
}
```

```
public void bar() {
      foo.toString;     // ERROR, lock is not held
      synchronized(lock) {
            foo.toString();// OK, the lock is held
      }
}
```

In addition to other objects, variables may be @GuardedBy("this"). They may be accessed inside synchronized blocks that are synchronized on `this`, or in methods that have the modifier `synchronized`.

The Lock type system prevents synchronization errors on shared resources, which makes it directly applicable to the security problem we want to prevent.

6.3 Converting to a Verigames type system

The Checker Inference Framework is built on top of the Checker Framework, and supports a type-inference mode in addition to the type-checking functionality of the Checker Framework. For type-inference, the Checker Inference Framework can keep track of additional information about the program beyond what the Checker Framework does. For example, it creates IDs for each variable and records constraints that are used to solve the type-inference. Constraints can be about relationships between variables (subtype, equality, etc.) or specific events in the program (assignments, field accesses, etc.), and every constraint contains context information about where the constraint was detected in the input program.

To turn a program into a game, Verigames needs type systems that are built on top of the Checker Inference Framework instead of the Checker Framework. The Visitor in a Verigames type system may specify when additional constraints are recorded. For an input program and Verigames type system, the Checker Inference Framework runs in inference mode. Then, the variables and constraints recorded by the Checker Inference Framework are processed by code called the GameSolver to determine the structure of pipes and boards in the game. Before the lock checker could be turned into a game, it was converted to a Verigames type system so that it uses the Checker Inference Framework instead of the Checker Framework directly. The Checker Inference Framework is built on top of the Checker Framework, so Verigames lock type systems can still be used for type-checking; it just supports an additional 'inference mode' that makes game creation possible.

6.4 Converting the Lock Type System to a Game

6.4.1 Challenges

The Lock type system is fairly simple to understand and use, but converting it into a game presented some unexpected challenges. It's simple to understand because the annotations express a very basic relationship. It's simple to use because the developer adding annotations understands the code, and all of the intended relationships between lock variables and locked variables.

In Verigames, we take unannotated code and a type system as input, and output a game. So far in Verigames, the type systems have had certain properties that make this task easier. First, each previous type system gives us a starting point for the properties of the program. In Nullness, we know that the null literal is always @Nullable, and that every dereferenced variable must be @NonNull. In Trusted, String literals and the methods in the stub file let us know what is @Trusted. These are hints into the intended properties of the overall system. We can encode these in the game, and the player just works to find an overall game configuration that minimizes conflict with this initial partial information about the program. This starting information essentially bootstraps the game.

Second, the properties expressed in the previous type systems involve only one variable at a time. A variable is simply @Nullable or @NonNull, for example, and the player can toggle each variable independently. The only relationship between variables is when the pipes flow into one another, and this is presented in a very simple and direct way to the player.

Third, every type system so far has a Supertype :> Subtype structure. The very model of pipes and widths was initially chosen to accommodate this type system structure well. For type systems with different structures, we do not fully understand how well the different structure will fit within the pipe game model.

The Lock type system has none of these properties; there is no concrete starting information in unannotated code, the relationship between variables is important and non-obvious, and the structure of the type system is new to Verigames.

What does the lock game look like for unannotated code? How do we find the concurrency properties of a program with just the source code's variables and synchronized blocks? We could use the synchronized blocks as a hint; whatever they are synchronized over must be a lock, and perhaps variables that are accessed inside are guarded by those locks. This leads to more questions than answers. Consider the following source code:

```
class Example {
  Object a;
  Object b;
  Object c;
  Object d;

  public method() {
    a.toString();
    b.toString();

    synchronized(c) {
      a.toString();
      b.toString();
    }

    synchronized(d) {
      a.toString();
      b.toString();
```

```
      }
    }
}
```

What can we infer from the synchronized blocks? Perhaps a and b are guarded by a lock, and the accesses outside the synchronized block are bugs. Perhaps a and b are not guarded by a lock, and the accesses inside the synchronized blocks are coincidence. If we say that a and b are locked, how do we know which is the correct lock for each variable? This is crucial, because the integrity of our type-checking results depend on getting the right relationships between the locks and the locked variables. Above, there is no proof that a and b are locked by one lock over another. Furthermore, trusting the synchronized blocks as the starting point of our analysis is insufficient because there may be missing or incorrect synchronized blocks. The Lock checker presents quite a few more challenges in converting it to a game than any previous Verigames type system. It has no concrete starting points of analysis in the source code, and the relationships between variables are complex and depend very much on the programmer's intention.

As a first approach, I decided to scale down the scope of the problem and investigate creating a game for a simpler problem in the Lock type system. I assume all of the locking relationships are annotated, and infer the @Holding method annotations.

6.4.2 The @Holding Game

The @Holding Game is designed for PipeJam classic, where boards represent methods, and method calls are shown as subboards within those boards. Each board has a pipe that represents the flow of control. Each lock has a unique colored star, which is placed on the flow of control pipe to indicate when the lock is held. Stars are added to the flow of control pipe at the beginning of a synchronized block, and subtracted at the end. Because it is perfectly legal for a method to have several synchronized blocks, this changes the model of the game slightly. It must be legal for non-starred pipes to flow into starred pipes within the same board. And, ideally, in future iterations the stars placed by synchronized blocks would be uneditable to the player because the game will not modify their placement in the source code and the player should focus on other tasks.

The only other necessary pipes are the locked variables, which should be narrow and gray because we don't want to change anything about their properties and there isn't a meaning for width in this game. When a variable is accessed, creates a Star-Test node, with the star color of the lock. If the flow of control pipe has that star, the output is narrow and flows into the next section of that variable's pipe. If the star is missing, the output is wide, and creates a conflict flowing into the next section of the variable's pipe.

The @Holding annotation on a method is represented by placing the indicated held lock's stars along the entire flow of control pipe for the corresponding board. Then, the @Holding method can only be used when the calling method is in also @Holding or in a synchronized block with the same lock. To ensure that the calling method has acquired the lock, we can say that non-starred pipes flowing into starred pipes across methods create a conflict.

Because this is a little confusing compared to the idea that it is allowed within a method, future iterations of the game could have some sort of visual indication of the difference. Perhaps there could be little star receptor ports at the beginning of the methods with @Holding, or something along those lines.

There is only one available player mechanic: adding and removing stars. A player can add stars to the flow of control pipe to resolve a Star-Test node conflict or an unmatched stars across boards conflict. This way, in the finished game the stars that go across the entire flow of control pipe in each method directly map to which locks the method is @Holding.

6.4.3 Implementation

The most interesting aspect of implementing the @Holding game was that it revealed just how much of our supposedly re-usable, generic code was actually still heavily influenced by the first type system structure, Supertype :> Subtype.

In the Checker Inference Framework, there were only existing constraints for pieces of code that were relevant to Nullness and Trusted. There's plenty of information about what happens in a program that the constraints don't record. I added constraints to record entering and exiting synchronized blocks, which are crucial to the @Holding game. Also, so far the Checker Inference Framework only has constraints for type annotations. I also needed to add a constraint that records method declarations, such as @Holding.

The GameSolver is the part of the Verigames pipeline that converts constraints from the Checker Inference Framework into a graph representation of a game, which will then be saved as XML to be loaded into the game-playing interface later. There is a superclass GameSolver, which handles all of the behavior and state that is agnostic to the specific type system. Each type system has its own GameSolver which overrides a small number of methods to add in the details of its specific behavior and any additional state. Without a serious refactoring that would have disrupted other developers working on the GameSolver for Nullness and Trusted, the @Holding GameSolver had to override almost every method of the 'generic' GameSolver. For example, the GameSolver creates a receiver pipe for every board, which has a negative variable ID and can therefor never have stars. The @Holding game does not need a receiver pipe, though it does need a flow of control pipe with the appropriate stars on every board. Additionally, the GameSolver creates getter and setter boards for each variable in order to use the correct width pipe when they are accessed. In the @Holding game, there is no need for getter and setter boards. When a locked variable is accessed, its value is not important and there is no meaning associated with its width. We only care if the associated lock is held or not. With many other examples like this, it's clear that the most generic idea of a GameSolver was a good intention, but the limited examples of type systems that influenced its design limited its application to new type systems in practice.

At the graph level, there are intentional design choices that work very well for the Verigames type systems so far, but are not ideal for the Lock type system. In the graph, pipes are either editable or not editable. It would be much better for the @Holding game if

there was higher precision in specifying how pipes can be edited, by separating width-editable and star-editable or even on a star-by-star basis. In another case, pipes with negative variables ID indicate that those pipes are not actually variables (such as this), and are not considered 'linked edge sets'. As a consequence, these non-variable pipes can never have stars because the graph only allows stars on 'linked edge sets'. To get around this, the @Holding game chooses a large positive number as the variable ID for the flow of control pipe, and just assumes that there will be no real variable in the program with a conflicting ID.

In the PipeJam game interface, there are plans to support a general Star-Test node but it is not yet implemented. The @Holding game currently uses a placeholder.

Here are some examples of code and the @Holding games they generate.

Figure 4: Accessing a variable within a synchronized block.

```
Object lock;
@GuardedBy("lock") Object locked;

public void method() {
  synchronized(lock){
    locked.toString();
  }
}
```

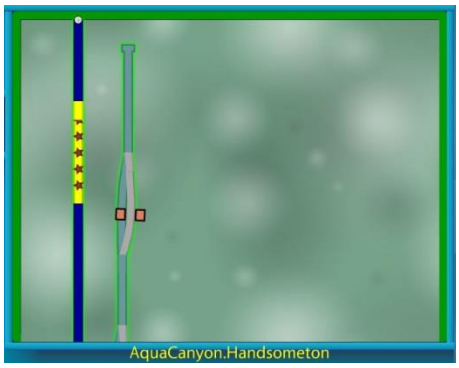


Figure 5: Accessing a variable within and outside a synchronized block.

```
Object lock;
@GuardedBy("lock") Object locked;

public void method(){
  synchronized(lock){
   locked.toString();
  }
  locked.toString()
}
```

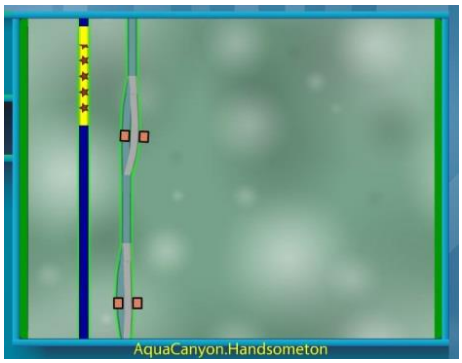


Figure6: A method with @Holding

```
@GuardedBy("lock") Object foo;

@Holding("lock")
public void bar(){
  foo.toString();
}
```

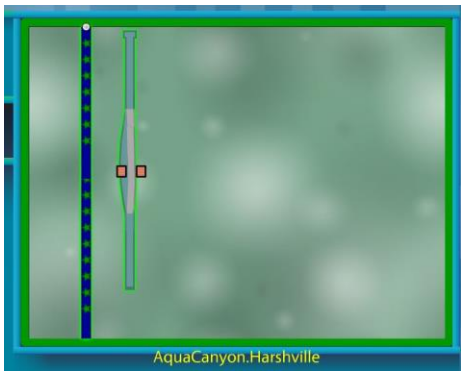
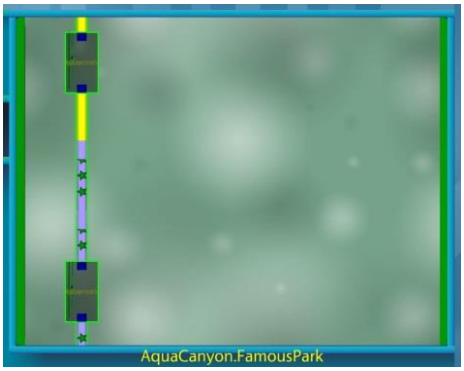


Figure 7: A method calling the @Holding method above, without and with the lock acquired.

```
public void method() {
  bar();

  synchronized(lock){
    bar();
  }
}
```



6.4.4 Lessons Learned

Importantly, when converting a new type system into a game, the structure and simplicity of the type system will dictate how easily it can be adapted to fit an existing game. If the structure hasn't been considered for a game design before, it will be harder to find a good mapping from code to game that works as an intuitive mental model. If the type system expresses more complex ideas, such as relationships between variables, then that also increases the complexity of the game-conversion task.

Another takeaway from this experience is fairly obvious, but is still worth saying because of how broadly it applies. It is extremely difficult to design a system that will correctly handle new problems and applications that were not fully understood or considered at design time. Every piece in the Verigames pipeline demonstrates this fact: the Checker Inference Framework, the GameSolver, the graph, and the game interface. This is an understandable and unavoidable fact of building software, and emphasizes that it is better to accept new challenges as they arise, learn from them, and iterate on a system design rather than attempt to get things perfect the first time.

7 Future Work

Verigames provides plenty of work that still needs to be done. For the Lock Checker, the @Holding inference game needs to be integrated with the gaming interface for the Star Test Node, pipe colorings, and other things. More interestingly, Verigames needs a more complex version of the game that fully infers the locking relationships needs to be considered. For Verigames as a whole, the remaining security properties must be expressed as type systems and turned into games. Hopefully this experience provides insights that will be valuable in turning other new type systems into games.

8 Conclusion

In this paper, I presented my work on Verigames, a system that lowers the skill level needed to formally verify code by presenting verification as a visual puzzle game that anyone can play. I discussed the @KeyFor relationship in the Nullness type system and my

experience detecting potential map and key pairs to be included in the game. I presented an extensible testing framework which tests for correctness at two crucial points in the Verigames pipeline. I explained my findings from the Trusted case study, where I annotated a real codebase with the trusted type system to better understand its strengths and weaknesses. I created 9 additional trusted type systems to help with Verigames' long term goal of verifying 25 security properties through games. I discussed the Lock type system, created a simple game for inferring the @Holding annotation, and described my observations about working with more complex and differently structured type systems in the current Verigames system. Though my work has been on quite varied problems and aspects within Verigames, it has all been with a goal of helping Verigames progress towards a full-fledged system that uses game play to lower barriers to formal verification of important properties.

9 References

1. "Verification Games: Making Verification Fun"
   Werner Dietl; Stephanie Dietzel; Michael D. Ernst; Nathaniel Mote; Brian Walker; Seth Cooper; Timothy Pavlik; Zoran Popović
   Proceedings for FTfJP 2012: The 14th Workshop on Formal Techniques for Java-Like Programs - Co-located with ECOOP 2012 and PLDI 2012, Papers Presented at the Workshop. 2012:42-49.

2. "Practical pluggable types for Java"
   by Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst.
   In ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis, (Seattle, WA, USA), July 22-24, 2008, pp. 201-212.

3. 2011 CWE/SANS Top 25 Most Dangerous Software Errors.
   http://cwe.mitre.org/top25/index.html

4. "Building and using pluggable type-checkers"
   by Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller.
   In ICSE'11, Proceedings of the 33rd International Conference on Software Engineering, (Waikiki, Hawaii, USA), May 25-27, 2011, pp. 681-690.
   http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011-abstract.html