Secure Autonomous Scripting

Brennon Brimhall, Marissa Angell
FRC 6844

March 5, 2018

1 Abstract

We detail a novel, rapid-prototyping approach to autonomous routine generation that leverages common programming languages and is corruption-proof and tamper-proof. This scripting system has been used effectively in 6844's inaugural robot. We detail the concept and source code below in the hopes that our work will be useful to other Java teams.[1]

2 Scripting for the Java Platform

Towards the end of 2006, a document detailing an update to the Java standard libraries was approved. This document was called JSR 223[2] and detailed a standard API whereby Java could interact with scripting languages, given an engine that interpreted the language and satisfied the API. In particular, this API allowed for specific Java objects to be passed into the script's context. This feature allows the scripts to call native Java methods on native Java objects. Similarly, the API allows Java to manipulate variables and call routines in the script.

So, how do you use it? Well, if you plan on creating a `ScriptEngine` that doesn't exist yet, you will need to write an engine that inherits from `javax.script.ScriptEngine` and put that on your `CLASSPATH`, generally via a JAR.[3] This isn't for the faint of heart, as writing an industrial-strength interpreter represents many man-hours of programming and debugging from those familiar with writing parsers, compilers, and the like – though one should never underestimate the power of smart people armed with free time. Luckily for us, Java bundled a JavaScript engine that uses those APIs.[4] Third parties have developed engines for many other languages,[5] most notably Jython.[6]

[1] Unfortunately, the implementation details are specific to Java, since the Java Runtime we have on the roboRIO dramatically simplifies making this all work. In theory, one could write a similar engine for C++. We're not certain if this could be done with LabView.

[2] See `https://www.jcp.org/en/jsr/detail?id=223` for more information.

[3] A JAR is a Java archive. It's a zipped folder with a bunch of compiled bytecode.

[4] That engine was named Rhino. Rhino was developed by Netscape in 1997 and was open-sourced in 1998. As of Java 8, Rhino was replaced with Nashorn. In 2016, a committee reviewed the status of the standard. Based on the vote of the committee, these features have been taken underneath the umbrella of the core JDK development team for Java 9 and onward.

[5] Over the course of several Google searches, we've concluded that implementations existed for Java (yes, that's pretty meta), Ruby, PHP, Scheme, Tcl, Groovy, AppleScript, and BeanShell. However, these projects seem to have been largely abandoned and we haven't found any code. Our opinion is that they shouldn't be used in FRC. Keep in mind that even Jython hasn't had a release in over two years.

[6] See `http://www.jython.org/`. They definitely support Python 2.7, but we're not aware of Python 3 support. This is contrary to the implications we find in our reading of documentation from the Python Software Foundation. If you know something I don't, please reach out.

```
import javax.script.*;

public class EvalFile {
    public static void main(String[] args) throws Exception {
        // USAGE: java EvalFile <Language> <File>
        // EXAMPLE: java EvalFile JavaScript hello.js

        // create a script engine manager
        ScriptEngineManager factory = new ScriptEngineManager();

        // create an engine; JavaScript is included, but you'll
        // need to ensure that you've got everything happy on
        // your CLASSPATH if you're working with something else.
        ScriptEngine engine = factory.getEngineByName(args[0]);

        // evaluate code from given file
        engine.eval(new java.io.FileReader(args[1]));
    }
}
```

Figure 1: EvalFile.java

```
print('Hello world!');
```

Figure 2: hello.js[7]

2.1 Basic Use

Using the scripting API is pretty simple. Assuming you have `EvalFile.java` and `hello.js` as detailed in Figure 1 and Figure 2, you'd just need to run `java EvalFile hello.js`, and you'd magically get `Hello world!` printed to the console.

2.2 Advanced usage with Nashorn

The scripting API supports the invocation of functions defined in the script as detailed in Figure 3 and Figure 4. These figures demonstrate loading an arbitrary file, parsing it, evaluating it as JavaScript, and calling its `hello()` function.

This permits us to load up JavaScript functions and call them later, and pass data from Java to them. This is where the scripting begins to be useful. This would be great to allow for modular, scriptable plugins, such as for logging purposes – or autonomous modes.

It turns out that we can pass Java objects to Javascript and call their methods, such as subsystems. We can also expose other variables to the context of the script, such as the game-specific string from FMS. Figures 5, 6, and 7 illustrate a practical example, where we're loading an autonomous

[7]Why are we using `print()` here instead of `console.log()`? It turns out that Nashorn is an implementation of ECMAScript (specifically ECMAScript 5.1 for the version bundled in Java 8), not strictly JavaScript. There's no Document Object Model (DOM) or other features typical of a web browser in Nashorn – that just doesn't make sense here. Thus, printing in Nashorn is done with `print()`, not `console.log()`.

```
import javax.script.*;

public class InvokeScriptFunctions {
    public static void main(String[] args) throws Exception {

        // Same as in the previous example
        ScriptEngineManager factory = new ScriptEngineManager();
        ScriptEngine engine = factory.getEngineByName("JavaScript");
        engine.eval(new java.io.FileReader("functions.js"));

        // Cast the engine to an invocable and call our functions:
        Invocable invocable = (Invocable) engine;
        invocable.invokeFunction("hello"); // Will print "Hello world!"
        System.out.println(invocable.invokeFunction("add", 3, 5));
    }
}
```

Figure 3: InvokeScriptFunctions.java

```
function hello() {
    print('Hello world!');
}

function add(x, y) {
    return x + y;
}
```

Figure 4: functions.js

```
import edu.wpi.first.wpilibj.Spark;

public class Drivetrain {

	Spark sparkLeft1, sparkLeft2, sparkRight1, sparkRight2;

	public Drivetrain() {
		sparkLeft1 = new Spark(0);
		sparkLeft2 = new Spark(1);
		sparkRight1 = new Spark(2);
		sparkRight2 = new Spark(3);

		sparkLeft1.setInverted(true);
		sparkLeft2.setInverted(true);
	}

	public void tankDrive(double left, double right) {
		sparkLeft1.set(left);
		sparkLeft2.set(left);
		sparkRight1.set(right);
		sparkRight2.set(right);
	}
}
```

Figure 5: Drivetrain.java

script and executing it on a roboRIO. We can expand on these concepts and create a full blown autonomous scripting system.

3 Trusting our Scripts

Before we get too excited about our newfound autonomous powers, there's some critical assumptions we need to acknowledge we're making:

1. We're trusting that the file we want actually exists. Just checking for files in a directory doesn't work either – that's just shifting the trust from a specific file to a specific directory.
2. We're trusting that the file is valid JavaScript, and that it doesn't throw any errors (such as dividing by zero).
3. We're further trusting that the JavaScript we're loading has a `periodic()` function that doesn't require any parameters.
4. While this may be at least partially covered by some of the assumptions we've made already, we're also trusting that this file hasn't been corrupted.[8]
5. We're trusting that we wrote this file. In theory, a malicious team could `ssh` into your roboRIO and modify the script. Hopefully this doesn't happen in practice.

[8]It's unlikely that a corrupted file will parse as valid JavaScript, for instance.

```
public class Robot extends TimedRobot {
    private Drivetrain drivetrain;
    private Invocable autonomous;

    public Robot() {
        this.drivetrain = new Drivetrain();

        // Load a JavaScript script engine.
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = factory.getEngineByName("JavaScript");

        // Bind the drivetrain symbol to the drivetrain object.
        engine.put("drivetrain", drivetrain);

        // Evaluate the file autonomous.js.
        engine.eval(new java.io.FileReader("autonomous.js"));

        // Cast this engine to an invocable.
        autonomous = (Invocable) engine;
    }

    @Override
    public void autonomousPeriodic() {
        engine.invoke("periodic");
    }
}
```

Figure 6: Robot.java

```
// A very simple autonomous routine.

function periodic() {

    // Just drive forward
    drivetrain.tankDrive(1, 1);
}
```

Figure 7: autonomous.js

Basically, it all this boils down to the idea that we're trusting this script. And that trust should be explicit, not implicit. We can do a better job of preempting issues by verifying our assumptions hold, as detailed below.

3.1 Verifying Script Existence

This one is pretty simple to pull off; Java is kind enough to throw a `FileNotFoundException` if our script doesn't exist. We need to make sure we catch the error and report this. A good way to proceed is to print a message and a stack trace. We can also make sure that we also say relay a message to the drivers via the Driver Station.

3.2 Verifying Script Correctness

This issue is much harder to verify; Java and JavaScript are Turing-complete languages. That means that any nontrivial analysis of their behavior is not computable.[9] There are two routes we can follow: we can have a human verify the script, or we can develop some tests that the script must pass.

It's critical to realize the implications of just trusting ourselves to verify correctness - we are relying on the script coming from us and relying us to catch errors in the script. This results in needing to certify the origin of the script and also place operational policies in place so bad scripts can't be deployed. That's a lot of work. And we can only trust our scripts as much as we trust our own ability to write good programs.

If partial confidence is acceptable, there is a lot we can automate. We do this by running a series of tests on our script to check for common pitfalls. For instance, we might check that the file parses, is semantically correct JavaScript, defines specific functions, etc. Realize that this means that we can only partially trust our scripts – we can only verify key features of our code.

Our opinion is that the best check of script correctness is a combination of human review and automated testing.

3.3 Verifying Script Integrity and Origin

There are two main (and related) ways of verifying a file's integrity. The first is to use a hashing algorithm to generate a fingerprint of the file. The idea here is that small changes in the file result in major changes to the hash, and that the hash is easy to compute for a given file. We can precompute the hash, store that in a file, compute the hash of the script at runtime, and compare them. It is extremely unlikely that you'll encounter a collision - where two files hash to the same fingerprint. The careful, mathematically-inclined reader will realize that generating a fingerprint of a larger file means that we're never going to get perfect confidence here. And they're right. However, the math lets us choose how confident we want to be. For a 256 bit hash, such as SHA-256, there's approximately $\frac{1}{2^{256}}$ chance for a collision, which means that there's approximately a $1 - \frac{1}{2^{256}}$ confidence level.

The second way borrows on the hashing idea of the first, and is known as a digital signature. In a digital signature, you are still hashing the file, but you're also using features of public-key cryptog-

[9] This may come as a shock, but not everything is computable; there are some problems that simply can't be solved by a computer. Per Rice's theorem, the problem of determining the behavior of an arbitrary program is an *undecidable* problem and is as difficult to compute as the halting problem. If it wasn't, then we'd live in a world that didn't have software bugs. That would be nice.

raphy.[10] A digital signature guarantees integrity, but it also guarantees that the file originated from the sender. The signature is secure because it is specific to the file; the algorithm produces different signatures based on the hash of the file.

A brief outline of how a digital signature might work in our autonomous scripting system is given below:

1. A team generates a public/private key set.
2. The team secures both the public and private keys. Your ability to determine the origin of the script is directly related to the security of the public and private key.
3. The team copies the public key onto the roboRIO, either in the form of a byte array compiled into the code or as a file.
4. The team secures the public key on the roboRIO such that it cannot be modified by a malicious actor.[11]
5. The team does not divulge the private key.[12]

Our suggestion is to compile in the public key into the JAR. This means that a malicious actor would have to decompile your bytecode to recover it, which is just as hard as replacing your code with their own.[13]

4 Use Cases

4.1 Rapid Autonomous Routine Iteration

That's a lot of theory, but we get some good applications. The impetus for developing this is that we're able to quickly iterate and prototype autonomous routines. Tweaking an autonomous routine does not require us to rebuild and redeploy code. The process of rebuilding and redeploying takes about 6-7 seconds via the `ant` script generated by the WPILib plugins for Eclipse. Our experience is that there's an additional 15-20 seconds of time that the roboRIO takes to be operational again – this includes reloading the CTRE Phoenix runtime, reloading the JVM, etc. This whole process takes about 20-25 seconds if you want to change a single line of code.[14]

[10] The exact way that a digital signature works is beyond the scope of this paper, but we encourage the interested reader to learn the specifics via the Internet. The Wikipedia article is pretty good.

[11] Those who know a bit about asymmetric cryptosystems might be a bit surprised to read that *both* the private and public key must be secured. Securing the private key makes sense; that's how you compute the signature to assign to a file. Why do you have to secure the public key? If anyone can modify the public key, then someone could come along and generate a new private/public key pair, replace the public key, and have the ability to execute arbitrary scripts that would pass verification. A public key may be world-readable, but must not be world-writable.

[12] In our opinion, the private key should only made available on an as-needed basis. Those who have the private key have the ability to execute arbitrary scripts on the roboRIO; they have deploy access. Our opinion is that it should be kept safe with a mentor so that a code review has to take place. If you want to get fancy, you could also implement Shamir's secret sharing scheme and distribute shadows of the key so that students could upload scripts in tandem without a mentor (or even to force a mentor to have to upload scripts in concert with a student).

[13] In reality, it would be best to secure the `lvuser` and `admin` accounts on the roboRIO by changing default passwords. In Linux, you can control file permissions via the `chmod` command. The trustworthiness of the scripts would then be related to the security of the root password. However, changing passwords has caused our team lots of grief – it appears that both the WPILib toolchain and the roboRIO's operating system requrie passwords be set to defaults. Changing our passwords required us to contact NI for assistance in reimaging the roboRIO.

[14] While the 6-7 second deploy from `ant` has been measured precisely (to the resolution of a second), we haven't measured the 15-25 second reload time. Thus the numbers for the reloading part are anecdotal, not empirical.

Instead of waiting 20-30 seconds each time we're reloading an autonomous routine, we're able to dynamically reload code on the fly as desired. Our experience is that calculating the signatures and `scp`ing the files over takes less than a second. Most of the time, the auto routine was modified and redeployed before the robot was wheeled back to the starting position.

4.2 Autonomous Code Sandboxing

One of our most dreaded fears is that a last-minute, untested edit to the robot source code results in some exception (probably a `NullPointerException`, if you're anything like us) that takes down the whole robot and leaves you unable to drive or move for a match.

Proper implementation of the principles detailed in section 3 helps us mitigate this problem. Autonomous scripts do not take down the whole robot programming, because they are sandboxed. They exist in a separate layer and only interact with our code across a strict barrier. Scripts may be poorly written and result in exceptions, but these exceptions only result in a loss of autonomous functionality, not teleoperated functionality.[15]

4.3 Dynamic Code

With the exception of some minor code that maps the operator interface to functionality, most of our code is related to the definition of subsystems. We're confident that we could expand the system to allow us to code most of the robot in JavaScript, and only use Java for defining our robot's subsystems if we desired.[16]

This is what the Python teams have been able to do for years. But we get the best of both worlds - we get the benefits of an officially-supported language and dynamic code.

5 Superiority over Alternative Autonomous Systems

There are several autonomous systems that have been published in the FRC community, such as 1114's AutoBuilder framework,[17] 1678's lemonscript,[18] RobotPy, and others. However, these systems do not meet the following criteria simultaneously:

1. Dynamically loaded at runtime. A failure to meet this means that a change in autonomous routines requires rebuilding and redeploying all the robot code. A more comprehensive treatment of this is found in section 4.1. Common systems that don't have this ability include the Command-based framework that WPILib provides, as well as 1114's AutoBuilder framework.

[15] One thing we're currently investigating is to run autonomous scripts when the robot is disabled. The robot shouldn't move, and we should learn if there's an issue with the script.

[16] We're further confident that we could expand the system to allow us to code nearly all of the robot in JavaScript, but we're not sure we're interested in going down this route. Instantiating Java objects in Nashorn is a bit of a pain, since you have to give it the fully-qualified Java class name; there doesn't seem to be a way to do the equivalent of an `import package.className` line. Alternatively, if you define everything in JavaScript, you're losing the type-safety that Java and C++ guarantees.

[17] AutoBuilder is available on BitBucket.

[18] Their code is available at `https://github.com/frc1678/robot-code-public/tree/master/c2017/lemonscript`.

2. It's Turing-complete; this means that the autonomous script can declare variables and access them at will. As such, the script retains full computational power.[19] As far as we can tell, lemonscript falls short here.[20]

3. It's secure. Other autonomous scripting systems appear to forgo a digital signature, corresponding digital signature verification, and testing for correctness. Even RobotPy, which satisfies the first two items, falls short here.

6 Source Code

The following is an edited excerpt from Borg, a library that we've been developing as a companion to WPILib. While we're not quite ready to recommend others use it, you can see it in it's full glory on GitHub.

```
package org.uvstem.borg;

import java.io.*;
import java.nio.file.Files;
import java.security.*;
import java.security.spec.*;
import java.util.*;
import javax.script.*;

/**
 * This class extends the WPILib TimedRobot class to provide Nashorn-based
 * autonomous scripting, and cryptographic verification of those autonomous
 * scripts.
 *
 * Note that your extension of this class must call super for the robotInit(),
 * autoInit(), autoPeriodic(), teleopInit(), and teleopPeriodic() methods if
 * overridden.
 */
public abstract class BorgRobot extends TimedRobot {
	private Map<String, BorgSubsystem> subsystems = new HashMap<>();

	protected PublicKey publicKey;
	protected SendableChooser<Invocable> autoModes = new SendableChooser<>();

	/**
	 * Run init() from the selected autonomous script.
	 */
	@Override
	public void autonomousInit() {
		Invocable i = autoModes.getSelected();
```

[19] Turing-completeness isn't necessarily something that's always a good thing in autonomous scripts, as that means that the interpreter is much more complex. However, keep in mind that we're using one that's pre-made by the folks who write the JDK and is battle-tested.

[20] It may be that other systems, like 1114's AutoBuilder and WPILib's command-based framework *could*, with significant massaging, be Turing-complete. (As an example of why we're couching behind words like *may* and *seem*, C++'s templating system has been shown to be Turing-complete, even though that wasn't necessarily the original intent.) We'll leave proving that those frameworks are Turing-complete to the Zebracorns.

```
}

/**
 * Run periodic() from the selected autonomous script.
 */
@Override
public void autonomousPeriodic() {
        Invocable i = autoModes.getSelected();
        try {
                i.invokeFunction("periodic");
        } catch (Exception e) {
            e.printStackTrace();
        }
}

/**
 * Initialize the public key for autonomous scripts and load them for use
 * via Shuffleboard.
 * @throws IOException
 * @throws InvalidKeySpecException
 * @throws NoSuchAlgorithmException
 * @throws SignatureException
 * @throws InvalidKeyException
 */
protected void initAutoScripts(byte[] publicKeyBytes, File scriptDirectory)
    ↪  throws NoSuchAlgorithmException, InvalidKeySpecException,
    ↪ IOException, InvalidKeyException, SignatureException {
        initPublicKey(publicKeyBytes);
        initAutoScripts(scriptDirectory);
}

/**
 * Initialize the public key for autonomous script signature verification.
 */
protected void initPublicKey(byte[] publicKeyBytes) throws IOException,
    ↪ NoSuchAlgorithmException, InvalidKeySpecException {
        X509EncodedKeySpec spec = new X509EncodedKeySpec(publicKeyBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        this.publicKey = kf.generatePublic(spec);
}

/**
 * Identify and verify potential scripts to use.
 * @param scriptDirectory The folder that contains the .sig and .js files.
 * @throws NoSuchAlgorithmException
 * @throws InvalidKeyException
 * @throws IOException
 * @throws SignatureException
 */
protected void initAutoScripts(File scriptDirectory) throws
    ↪ NoSuchAlgorithmException, InvalidKeyException, SignatureException,
```

```
    ↪ IOException {
        Set<File> signatures = new HashSet<>(Arrays.asList(scriptDirectory.
            ↪ listFiles((File dir, String name) -> {
                if (name.contains(".sig")) {
                        return true;
                }

                return false;
        })));

        //Verify scripts
        Signature publicSignature = Signature.getInstance("SHA256withRSA");
        publicSignature.initVerify(this.publicKey);

        ScriptEngineManager manager = new ScriptEngineManager();

        for (File sig : signatures) {
                String scriptName = sig.getName().replaceFirst("\\.sig", "\\.
                    ↪ js");

                File[] scriptMatches = scriptDirectory.listFiles((File dir,
                    ↪ String name) -> {
                        if (name.equals(scriptName)) {
                                return true;
                        }

                        return false;
                });

                if (scriptMatches.length == 1) {
                        publicSignature.update(Files.readAllBytes(
                            ↪ scriptMatches[0].toPath()));

                        if (publicSignature.verify(Files.readAllBytes(sig.
                            ↪ toPath()))) {
                                ScriptEngine engine = manager.getEngineByName
                                    ↪ ("nashorn");
                                try {
                                        setUpScriptContext(engine);
                                        engine.eval(new FileReader(
                                            ↪ scriptMatches[0]));
                                        autoModes.addDefault(scriptName, (
                                            ↪ Invocable) engine);
                                }
                        }
                }
        }

        SmartDashboard.putData("Autonomous modes", autoModes);
}
```

```
        /**
         * Expose each registered subsystem to the autonomous script engine.
         * @param engine
         */
        private void setUpScriptContext(ScriptEngine engine) {
                engine.put("gameData", DriverStation.getInstance().
                    ↪ getGameSpecificMessage());

                for (String subsystem : subsystems.keySet()) {
                        engine.put(subsystem, subsystems.get(subsystem));
                }
        }
}
```

7 Acknowledgements

We appreciate the following individuals for reviewing this paper prior to publication and would like to publicly thank them: Matthew Evans, Jacob Dean, Brian Maher, Audrey McMahon, Paul Sargunas, Carl Springli, Justin Tervay, and Max Westwater.

8 Errors and Feedback

While we've taken the precaution of having multiple people in the FRC community proofread and review our paper for clarity and accuracy, we're not perfect. There's bound to be an error in here somewhere. As such, we make no claim about the fitness of any code or concept we've discussed for any purpose.

Should you be smarter than us and notice an error in this whitepaper, please reach out us via email at `programming@provotypes.org`, via Chief Delphi direct message, or via a post on the corresponding Chief Delphi thread. Similarly, reach out along those channels if you'd like to discuss something related to this paper offline.