# Chapter 4.2—Languages and Security: Safer Software Through Language and Compiler Techniques

Rahul Simha and Scotty Smith

Department of Computer Science, The George Washington University, Washington, DC

**Keywords:** Compilers, Runtime Systems, Static Analysis, Code Integrity, Watermarking, Obfuscation

**Abstract.** Embedded systems, such as those found in mobile phones or satellites, have grown in popularity in the recent years. Code that executes in these environments need to be verified as safe, so they do not expose sensitive data or hidden APIs to the outside world. With enough knowledge of the code and then environment in which it executes, malicious entities can find and exploit vulnerabilities for their own gain. Failure to protect and verify executing software can leak or corrupt sensitive data, and in extreme cases cause loss of the device. This chapter explores security through language, compiler and software techniques. The techniques and discussion apply to general system security. However, they are equally applicable to the systems described above.

## 1 Introduction

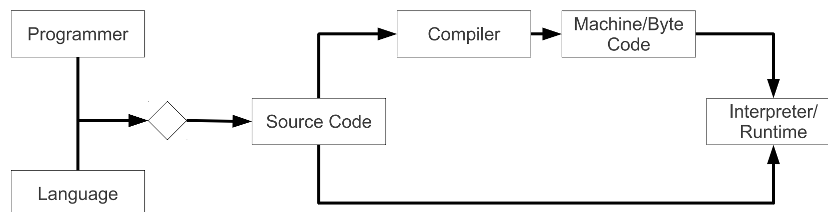


**Fig. 1.** Software Development Cycle

Developing secure software is a daunting task. Many tools and techniques have been created to alleviate the pressure of writing secure and efficient code. Some techniques were initially developed in the 1960's, while others

are more recent developments. In the following chapter, we will be discussing defenses for different attacks on code and software systems. The defenses in this chapter will fall withing the framework seen in Figure 1. We discuss *Watermarking* in Section 2, which defends against software piracy, and can occur in the source or machine codes, or even during the runtime system. *Obfuscation* in Section 3 defends against reverse engineering, or act as a form of DRM. These techniques work at a similar level to watermarking. In Section 4, we discuss *Code Integrity* techniques to assure that trusted code is executed correctly. These generally occur during runtime, but elements can be setup during the compilation phase. *Proof-carrying code* (PCC) in Section 5 covers a large portion of the framework seen in Figure 1, in an attempt to add code to the trusted computing base. Section 6 discusses *Static Analysis Techniques* for software protection occurring prior to execution, classically in the compilation phase. *Information Flow Techniques* in Section 7 discusses how prevent invalid data access. Tools for these techniques can operate statically on the source code, or dynamically during the runtime of the program. Section 8, several tools for software verification are discussed, operating on source code or at system runtime. Finally, Section 9 discusses *Language Modifications* to increase software security.

# 2 Compiler Techniques for Copyrights and Watermarking

Software piracy is a drain on commerce that is increasing due to the ubiquity of Internet use. Global piracy rates of 43% were reported in 2009, a 2% increase from 2008 [14]. All signs point toward piracy as a growing trend, as the average knowledge of Internet users increases. Many techniques have been tried to prevent piracy, such as the Content Scrambling System used for DVD's [41], Apple's FairPlay DRM for

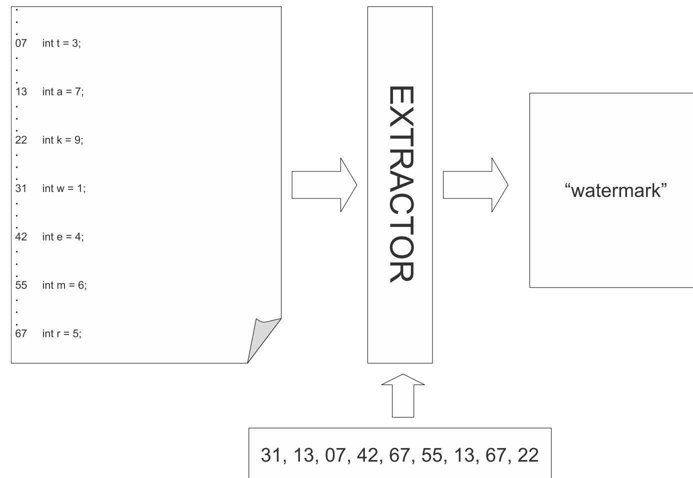


**Fig. 2.** Watermark Example

downloadable music [21] and the SecurROM DRM scheme for computer games [34]. These techniques have achieved varying degrees of success.

Watermarking is a technique that was developed in the $13^{th}$ century that has been adapted as a way to protect digital goods. Watermarks on physical goods are often used to identify counterfeits, as they will have an incorrect or missing watermark. Digital watermarks can work in the same way; a content creator can embed information that cannot be perceived unless the correct "filter" is used. Counterfeit content can be identified if the correct "filter" does not recover the correct watermark. Many digital goods can be protected using watermarks, such as digital photos, documents, digital music, and computer software.

In the following section, we will focus on watermarks for executable software. We will introduce static and dynamic watermarks, and the tradeoffs between them. We will discuss the types of attacks a watermark must defend against, and how attacks disrupt the retrieval of a watermark. Finally, we will review what current research has been able to accomplish in the area of software watermarking.

## 2.1 Watermark Basics

While there are many watermarking techniques of varying complexity, they all follow the same rule: insert some information that does not change the overall behavior of the program, but is recoverable so that the program can be identified. This can be done in software anywhere from the high-level source code down to the binary.

A software watermarking system is defined by three functions. First, an embedding function, which takes a program $P$, a watermark $w$ and a key $k$ and transforms $P$ into $P_w$, which has the same semantics as $P$ with $w$ embedded in it. Second is an extraction function, which takes a program $P_w$ and a key $k$. with this information it is able to extract the watermark $w$ from $P_w$. Finally there is a recognizer function, which takes a program $P_w$, a key $k$ and a watermark $w$. The recognizer function returns a confidence level of whether $w$ is contained in $P_w$ (a percentage).

The watermarks can be used to accomplish several things. As with a normal watermark, a piece of watermarked software can be used to authenticate the original author. A program creator can embed a watermark to prove they are the original author. Anyone who would wish to fool someone into thinking they wrote the software would need to remove the watermark first. A software watermark can also be used as a way to track the original purchaser of a piece of software. The embedded watermark can contain identifying information of the purchaser. If someone else is found with a copy of the program with another persons watermark, then it is not a legitimate copy. Again, the watermark would have to be removed to fool the system.

There are two main classes of watermarks: static and dynamic. Static watermarks are those that can be extracted directly from the program without execution. The simplest form of a static watermark is a copyright string in the code's comments, or an actual string variable in the program. Of course, such a watermark is is easily identified and removed.

Thus, it is necessary to hide a watermark to achieve full protection. More sophisticated static watermarks involve permuting code or using register allocations to encode the watermark.

Dynamic watermarks usually require program execution in order to generate or extract the watermark. For example, a specific set of commands could be used to generate the watermark in a different part of the program (which then needs to be extracted). The benefit of dynamic watermarks over their static counterparts is that typical transforms used to break the static watermarks typically have no effect on the dynamic watermarks. Similar to the static watermarks, the dynamic watermark needs to be hidden. If the watermark generator is easily identified, then an attacker can remove or replace the watermark.

## 2.2 Attacks on Watermarks

As discussed above, there are attacks that target watermark systems. There are more attacks possible than those described below, but a good watermarking system should strive to be resilient to some of these attacks:

- **Additive Attack** In an additive attack, the attacker inserts a custom watermark into the program. In doing so, the attacker casts doubt on the validity of the original watermark.
- **Subtractive Attack** In a subtractive attack, the attacker removes most (if not all) of the original watermark, such that recovery is incomplete or impossible.
- **Distortive Attack** In a distortive attack, transformations are applied to the program to prevent the recovery of the watermark by scrambling the locations where the watermark is expected.

In the additive attack, the only change is a parameter to the recognition function. For both recognitions, the function is the same, and so is the key. However, both watermarks are recognized successfully. This leads to

confusion over which is the correct watermark, as both watermarks are viewed as correct. For the subtractive attack, the attacker needs to identify what pieces of code are part of the watermark, and must remove the watermark without altering the behavior of the program. If the attacker is unable to preserve program behavior, then being able to eliminate the watermark is useless, as the program will no longer function correctly. Distortive attacks can eliminate the issues experienced in subtractive attacks when applying semantics-preserving transformations. However, as we will see in section 3, transformations will likely increase the size of the program, and potentially the runtime of the program.

## 2.3 Current Research

The software distribution model is generally more complex than the producer/consumer dictates. A watermark applied by an author to a program is only recoverable by using information provided by the author. In more complex, yet common, distribution models, the author allows one or more distributors access to the program. Each distributor then sells the program to the user. While the watermark applied by the author can protect against plagiarism, it may not be sufficient as a DRM technique.

Christian Collberg et. al. have been working on a solution for watermarking in Java applications[44]. The resulting watermarking technique is referred to as a *semi-dynamic* watermark. It is semi-dynamic because the extraction of the watermark does not need to execute the entire program, but the extractor does have to run a reconstructed program that is assembled from pieces of the original program.
Embedding the watermarks in this scheme rely on the identification of articulation points in the Control Flow Graph (CFG). Articulation points in the CFG are nodes which if removed disconnects the graph. A new basic block is created to encode each watermark. An articulation point

is chosen, and split. The new basic block is inserted into the control flow graph at the articulation point. This is done in such a way that the new basic block does not become an articulation point (by connecting the new basic block to both of the split articulation points). This ensures there will never be a collision between the series of watermarks. This allows for multiple watermarks to be embedded without corrupting their recovery.

Extracting the watermark requires the extractor to determine whether or not a watermark is embedded in the basic block. The basic block needs to have at least one articulation point incident to it, and it needs to contain certain instructions, which indicate that it stores some watermark data. These basic blocks are used to construct new methods that are combined into a new program. This new program is executed by the extractor program to recover the watermark.

Experiments with the semi-dynamic multiple watermarking algorithm show that it can embed and recover several watermarks, which is difficult in other watermarking techniques. The performance of the technique is comparable to the graph-theoretic watermarking algorithm [40], though there are situations in which the performance is worse. Unfortunately, the current state of the algorithm is not resistant to manual subtractive, and several additive and distortive techniques.

# 3 Compiler Techniques for Code Obfuscation [8]

Obfuscation started not as a means for securing software, but as a thought experiment as to how unintelligible a fairly simple piece of code could become. To that end, in 1984, the first *International Obfuscated C Code Contest* (IOCCC) [30] was held in an attempt to find such code.

This annual contest is held to find the best obfuscated C code each year, and with every year the entries become more complex.

The following section deals not with the entertainment factor of obfuscated code, but rather the implications of such code. As we will see in the following sections, code obfuscation can be both a help and a hindrance. The techniques developed for code obfuscation can often be used to hide malicious code in otherwise innocuous programs.

## 3.1 What is Obfuscation

Obfuscation is a technique by which code is transformed into a form that is semantically the same as the original program, but is difficult to extract meaning from. What "difficult" means in the previous definition is left up for interpretation. Under the above definition, many typical techniques qualify as being an obfuscation. For example, compilation is a form of obfuscation, as it converts source code into another form (e.g., machine code, byte code). There do exist programs that will "decompile" the compiled code back to source code, but in most cases it will only be vaguely similar to the original source.

There are four main classes of transforms used to obfuscate a piece of code:

1. **Abstraction Transformations** alter the structure of a program by removing program information from the functions, objects, and other source-level abstractions,
2. **Data Transformations** replace data structures with other data structures that reveal less information about the stored data.
3. **Control Transformations** alter the control flow structure of the program to hide the execution path.
4. **Dynamic Transformations** insert code into the program that causes the program to be transformed during execution.

There is a significant trade-off when determining how much obfuscation to use. Some of the above transformations can greatly increase code size, which in turn can lead to increases in execution time. If not enough obfuscation is used, then it may be straightforward to undo the obfuscation. The correct amount of obfuscation depends on how much performance you are willing to sacrifice, and how much protection is necessary to achieve with the obfuscations.

## 3.2 Applications of Obfuscation [8]

One of the major uses of obfuscation is in reverse engineering prevention. Using the control and abstraction transformations, the control flow and structure of the program can be hidden. This will make it difficult for a user to determine what classes are being called, to the point that the actual code being executed is hidden. Adding in the dynamic transforms can greatly increase the uncertainty in the executed code.

Unfortunately, obfuscation is a double-edged sword. Software developers can use this to protect their code from being reverse engineered, but so can malware developers. As a matter of fact, several virus attacks of the past decade have used obfuscation techniques to hide their attacks. These techniques, especially the dynamic transforms, make it difficult for anti-virus software techniques to catch such attacks.

Obfuscation can be used for DRM as well. The techniques described above can be used to generate a unique executable for each purchased piece of software. This unique executable can be used to trace a pirated piece of software back to the original owner, who can then be held accountable. The diversity in the code generated in this way can have an additional benefit: bugs and vulnerabilities in the code that depend on certain control flow sequences can be mitigated. Given a vulnerability in one instance of the program, it is non-trivial to exploit the same vulnerability in a different instance of the same program.

**Listing 1.1.** Account Example

```
1   public class Account{
2       private static int account_number_seed = 0;
3       private String first_name, last_name;
4       private long account_number;
5       private double balance;

7       public Account(String f_name, String l_name){
8           first_name = f_name;
9           last_name = l_name;
10          balance = 0;
11          account_number = account_number_seed++;
12      }

14      public double getBalance(){
15          return balance;
16      }

18      public void deposit(double amount){
19          balance += amount;
20      }

22      public void withdraw(double amount){
23          if(amount > balance){
24              System.out.println("ERROR: balance is too low "
25                                 +"to process transaction");
26          }else{
27              balance -= amount;
28          }
29      }

31      public String toString(){
32          return "First Name: " + first_name +
33              "\tLast Name: " + last_name +
34              "\nAccount Number: " + account_number +
35              "\nBalance: " + balance + "\n";
36      }
37  }
```

**Listing 1.2.** Ofuscated Example

```
1   public class A1{
2       private static int a = 0;
3       private String f, l;
4       private long n;
5       private double b;

7       public A1(String f, String l){
8           this.f = f;
9           this.l = l;
10          b = 0;
11          n = a++;
12      }

14      public double b(){
15          return b;
16      }

18      public void d(double a){
19          b += a;
20      }

22      public void w(double a){
23          if(a > b){
24              System.out.println(U.d("REEBE: onynapr vf gbb ybj ")
25                                 + U.d("gb cebprff genafnpgvba"));
26          }else{
27              b -= a;
28          }
29      }

31      public String toString(){
32          return U.d("Svefg Anzr: ") + f +
33              U.d("\tYnfg Anzr: ") + l +
34              U.d("\nNppbhag Ahzore: ") + n +
35              U.d("\nOnynapr: ") + b + "\n";
36      }
37  }
```

### 3.3 Transforms

**Abstraction Transformations** Consider the Java class in Listing 1.1. The programmer who wrote this code created this class for a specific purpose: to represent a bank account in the overall program. The class name, method names, and variables leak information and intent. The fact that this data was broken into its own class leaks some information as well. The goal of an abstraction transformation is to remove some information that source abstractions leak.

Listing 1.2 shows a simple abstraction transform (performed manually for this example). All of the identifiers from the original program have been replaced with unrelated (and significantly smaller) identifiers. Similarly, all of the String literals in the program have been encrypted in order to prevent information leakage about the original source code. It is more difficult to extract meaning from the code in Listing 1.2, as compared to the original code.

**Data Transformations** Even after applying the above transforms to the code, an attacker may still reason about where to look for data, should they already know what the class does. This is because they know the *type* of data they are looking for, should it be an integer, string, or some known data structure. Data transformations combine data structures and create new data structures that contain superfluous data fields, in order to obscure the true meaning behind the data structure.

**Control Transformations** Control transforms alter the control flow of the program to further hide the sequence of commands that are executed in the program. A simple version of this kind of transform can be achieved in the code in Listing 1.2 by condensing all of the code into one method. The code blocks are separated by some conditional control flow structure such as a switch (see Listing 1.3). Now there are only two methods in the class, where there used to be four. As a result, any calls to the hidden three methods now go through one method header, with a control switch to determine which piece of the method to execute. This is a fairly weak obfuscation, as it can be fairly easily reversed, but more powerful control transforms exist.

**Listing 1.3.** Control Example

```
1  public double o(double a, int c){
2      double r = -1;
3      switch(c){
4      case 0:
5          r = b;
6          break;
7      case 1:
8          b += a;
9          break;
10     case 2:
11         if(a > b){
12             System.out.println(U.d("REEBE: onynapr vf gbb ybj ")
13                                + U.d("gb cebprff genafnpgvba"));
14         }else{
15             b -= a;
16         }
17         break;
18     default:
19     }
20 }
```

**Dynamic Transforms** With the above static transforms, it is still possible to work around the obfuscations to get to the actual code being executed. Every time the program is run, the same commands are executed, so determining what pieces of code correspond to certain aspects of the program is a straightforward (but often time consuming) process of executing the program over and over with different inputs. The data and control flow can then be traced to figure out the true behavior of the program.

Since dynamic transforms are performed at runtime, they do not necessarily have the same problem. A dynamic transformation can cause the control flow of the program (and indirectly the data flow) to evolve over time. In some cases even successive execution on different inputs can result in different paths through the program. The major drawback of a dynamic transform is additional runtime required to perform the transforms. This is the main reason why most dynamic transformations are not simply the static transformations coded to be performed at runtime, as they are not the quickest of algorithms.

### 3.4 Current Research

For every new transformation or technique, an argument needs to be made as to how well it achieves the obfuscation goals. One thing that has been missing from the field is a way to quantitatively evaluate transformations on how well they hide the original program's intent. Such a framework would work not only as a way to test new obfuscation techniques, but could also be used to compare current obfuscation techniques against one another.

Koen De Bosschere et. al. started building such a framework [1]. They describe a framework that operates using benchmarking ideas from other areas such as hardware design. Other metrics for evaluating obfuscation existed, but none had been widely accepted by the obfuscation commu-

nity. This work utilizes the Software Complexity Metrics developed in the 1970s and 1980s. These metrics, originally intended to describe the complexity of a computer program in a software engineering sense, are used to determine the *change* in complexity of an obfuscated program.

Koen De Bosschere has also worked on building tools to aid in the understanding of obfuscated code. De Bosschere et. al. have developed the Loco toolkit [24], a graphical framework that can interactively obfuscate and deobfuscate programs. Loco is built upon the DIABLO framework, also developed in part by De Bosschere, and is the obfuscation underpinning of the Loco toolkit.

The main goal of the Loco toolkit is to provide an environment for programmers to experiment with and evaluate obfuscation techniques. Obfuscations can be applied automatically or manually. The results of these obfuscations can be observed via the control flow graphs displayed by the toolkit. The tool also gives the user the ability to obfuscate on a finer-grained level than is provided by typical obfuscation software, which usually operates on the code as a whole. In the Loco toolkit, the user can specify which parts of the code to obfuscate, leaving unselected code unaltered.

One of the major uses of the Loco toolkit, as proposed by De Bosschere et. al. [25], is to provide a way to train users to understand obfuscated code. The crux here is that Loco can automatically perform some obfuscations/deobfuscations, teaching an analyst how to develop new obfuscations or how to extract meaning from obfuscated code. Once an analyst learns these techniques, they can extend the framework to meet their needs, to the benefit of others using the Loco toolkit.

Another of the issues faced with code obfuscation is the need to avoid detection. Often it can be helpful to hide that the program was obfuscated. Malicious software tries to achieve this, and thus it is important to understand the techniques that malware providers use to achieve this

goal. For many obfuscation techniques, statistical and semantic analysis techniques are able to extract the meaning from the obfuscated code.

A team of researchers at the College of William and Mary, led by Dr. Haining Wang, have developed a novel technique for solving this problem [42]. Their technique utilizes the steganographic technique of mimic functions. Mimic functions were developed in 1992 by Peter Wayner, and are designed to convert data into an output that *mimics* the statistical properties of some other data set. The main idea behind mimic functions is to develop Huffman trees that can be used in reverse. The Huffman tree decode phase is used as the encode phase of the mimic function. Then, when the file is ready to be interpreted, the encode phase of the Huffman tree is used to retrieve the original file.

The Mimimorphic engine takes, as input, a set of target programs to mimic, and a program to transform. The programs to mimic should be a trusted program, such as a word processor or web browser. The target programs are converted into a form of Huffman tree (forest), which can then be used to create a new program that resembles the trusted program set. The generated program can be executed, but has no semantic meaning. In order to be executed correctly, the program must be decoded, which requires the mimicry data computed for the encoding phase. This data is currently embedded in the executable, which greatly increases the program size.

Extracting a fingerprint for the decoder is usually the weakness of polymorphically obfuscated malware (malware obfuscated using dynamic transforms), because the decoder is the only portion of the malware that is executable. However, since the mimimorphic scheme is executable, it can better hide the decoder, increasing the likelihood that the decoder will not be found. Current research is working towards blending the control flow of the obfuscated binary with that of the decoder, further hiding the decoder in the payload. Results from the current prototype are promis-

ing, with binaries translated using the mimimorphic engine generate several fingerprints that also match “trusted” programs.

## 4 Compiler Techniques for Code Integrity

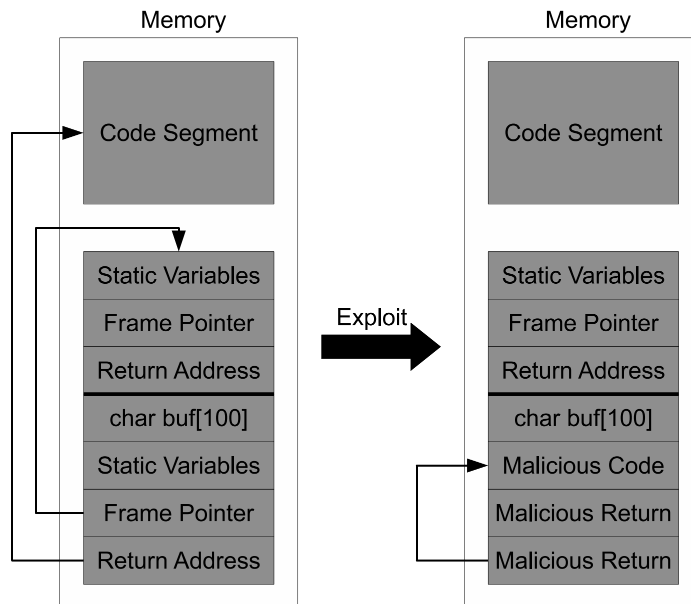


**Fig. 3.** Overview of a generic buffer overflow attack

In August of 2003, the Blaster worm began its attack on Windows-based computing systems. Blaster is similar to the Code Red worm in 2001. Both worms attempted to perform a distributed denial of service (DDOS) attack on various websites. These attacks exploit a well understood vulnerability known as a **buffer overflow**.

A buffer overflow occurs when data is written outside of the bounds of its allocated memory buffer. Stack memory, which holds automatic variables and function return addresses, is a common target for buffer overflow attacks. Overflowing a stack-allocated buffer enables an attacker

to overwrite the return address. If the return address is overwritten, the return from that function no longer preserves the application's original control flow, instead branching to an arbitrary location determined by the attacker. If the attacker supplies the address of a buffer containing malicious code, then the CPU will execute the injected malicious code with the same privilege level as the original application.

Buffer overflow attacks can be categorized based on where overflow data is written and how the attack vector affects the system. These categories are stack smashing, arc injection, pointer subterfuge, and heap smashing [32]. The canonical buffer overflow attack is a stack smashing attack that was written by a hacker named "Aleph One [31]." The attack is illustrated in Figure 3. In this example, a statically declared buffer ("buff") on the stack is the target of the attack. This buffer has a fixed size of 100 bytes. However, several *C* library functions do not check the size of the buffer that they write to. For example, the *strcpy* function will copy the entire source buffer into the destination. An intelligent hacker can create a string that will overflow the buffer, forcing a function like strcpy to write beyond the 100 bytes and overwrite the return address of the current stack entry. When the function returns, the hacker's new return address is used, allowing the malicious code to execute.

A simple defense mechanism is to prevent execution of data located in the stack memory region, thus preventing the injected code from executing. This is easily accomplished with non-executable memory regions, for example by using memory pages with no-execute (NX) permission bit or a Harvard architecture, in which data and instruction memory are explicitly separated. In response to such defense techniques, attackers have devised more complex attacks known as "return-to-libc", "return-oriented programming" [35, 5] or architecture-specific attacks [17]. In the return-to-libc attack, attackers overwrite the return address with the location of a C library function, such as `exec()`, or `system()`. These func-

tions accept parameters such as "/bin/sh", which will spawn a shell with the same privilege as the exploited application. A successful attack can potentially give the attacker the ability to run any commands on the vulnerable system with root privilege. Return-oriented programming involves creating a forged stack with a series of fake callers by injecting return addresses along with parameters to the stack. Thus, the attacker creates a sequence of operations that are executed by existing code based on the contents of the forged stack. Preventing the more complex variants of the stack smashing attack requires preventing the return address from being successfully overwritten, since the attack code is no longer injected onto the stack.

The buffer overflow attack does not need to occur on the stack, however. Recent work shows that the heap is as vulnerable to overflow vulnerabilities as the stack [22]. Using techniques similar to those used in the stack-based buffer overflow attack, a hacker can overwrite heap metadata, corrupting or recovering user data. Function pointers can be overwritten to allow arbitrary code to be executed on the system, potentially giving full control to the malicious entity.

In this section, we will discuss ways to combat these and other attacks. Both hardware and software mechanisms to protect against buffer overflow exist, but we will focus solely on the language/software side. See Chapter 4.1 for a review of hardware protection mechanisms. Although much of this section will focus on $C/C++$, other languages are vulnerable to these attacks.

## 4.1 Current Research

The reason that stack-based buffer overflow attacks succeed is because the return address of some function is undetectably altered to point to malicious code. One way to combat this issue is by inserting some markers around the return pointers that can be used as validation points

to detect changes to the return address. This is the main contribution of the StackGuard team [10].

StackGuard is designed as a patch for `gcc`. It alters the `function_prologue` and `function_epilogue` functions in `gcc`. The `function_prologue` function is responsible for pushing the protection marker (called "canary") onto the stack. This canary is a randomized value inserted between the return address and the rest of the local function variables. The `function_epilogue` checks to make sure the canary word is unaltered before the program returns to the callee. A typical buffer overflow attack now needs to alter the return address while leaving the canary intact in order to avoid detection. Ensuring that the canary word is sufficiently hard to determine helps, but being able to protect the stack without relying on canaries is desired.

StackGuard can use the MemGuard tool to detect overflows without canaries. MemGuard is intended to help code debugging [9], but the tool has the side effect of being able to detect changes to certain regions of memory. Again, this version of StackGuard is a patch for `gcc`, altering the same functions as the canary word version. This version does require the MemGuard tool to be running, as MemGuard becomes the only way to write to certain memory regions that have been designated read-only.

StackGuard is not the only software technique that can prevent buffer overflow attacks. Another approach uses what is known as a *shadow stack* to protect the return addresses. The Transparent Runtime Shadow Stack [38] is a runtime system built on a dynamic binary rewriting framework, and modifies how return addresses are handled. It operates on the same idea behind the compiler oriented StackShield [39]. The idea is that return addresses are stored separately from the stack frame. The return address is still part of the stack frame, but a copy is stored in a different stack. On a function return, the entries on the real stack and the

shadow stack are compared. If the two values do not match, an exception is thrown, preventing the buffer overflow attack from succeeding.

In the heap-based attack mentioned earlier in this section, one of the main causes of the vulnerability is the fact that program data and heap metadata are stored contiguously. However, separating the metadata from the program data is not sufficient, as an attacker with knowledge of the standard heap allocation techniques can predict where certain objects will get allocated. This knowledge allows the attacker to overwrite specific locations in the heap, which is sufficient to undertake a heap-based attack.

Heap Server[22] is a project designed to protect the heap from such attacks. Heap Server is a software process that manages the heap for other processes. The only modification necessary for the host language/process is in the allocation/deallocation routines. Heap Server manages the heap metadata separately from the program data, but uses clever storage mechanisms so that the lookup overhead for heap data is minimized. One of the interesting caveats of using Heap Server is that all of the heap data is stored in a completely separated protection domain. This means that any access or modification to the heap has to undergo inter-process communication (IPC). Luckily, the overhead of IPC can be hidden through clever optimizations.

Another protection mechanism provided by Heap Server is obfuscation. In order to prevent attackers from being able to predict where important elements of the heap are located, Heap Server randomizes dynamic allocations. This is achieved with two mechanisms. First is the insertion of random amounts of padding between heap elements, and second is selection of a random element from the free heap blocks to allocate for data. This allows for a program that is executed multiple times to have different heap layouts, minimizing the chance for an attacker to determine where critical heap data is stored.

# 5 Proof-Carrying Code and Authentication

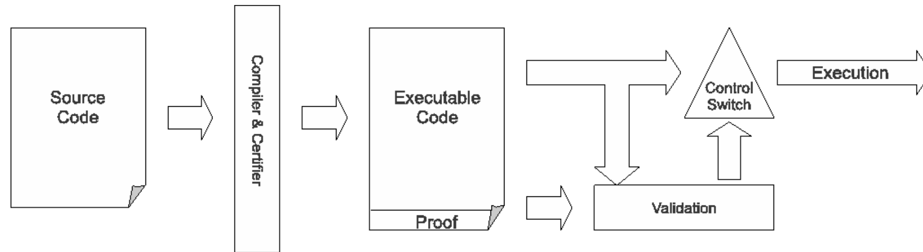


**Fig. 4.** Overview of Proof Carrying Code

As described in section 3, it can be difficult to ascertain at compile time what a program will do. So how can we increase our confidence in the software we are using beyond the simple "trust the vendor"? One such technique is to use *run-time contracts*, such as the ones employed by the Android OS with third party applications. When installing new applications, the installer notifies the end user on what the application will do and what information the application will access. These permissions are statically defined in the application. This solution is fine for the Android OS, as all programs are sandboxed and a runtime monitor validates the contracts. For general computing platforms, such a mechanism is more difficult to implement.

One solution for verifying a program's behavior is known as Proof-Carrying Code (PCC)[27]. In PCC, during the compilation phase, a proof is constructed detailing what the program will do and access. This proof is stored as part of the program's binary. When an end user executes this binary, the provided proof must be verified before the system begins execution (see figure 4). Instead of using a complex analyzer on the executable, a simple verifier can ascertain if the executable code and the compiler-generated proof are consistent.

PCC has several complications. For example, how does one generate the proof? Proofs can be generated by a compiler, or they can be constructed by the developer. There has to be some agreed upon definition of what operations the program is permitted to perform, and what operations are considered “bad”. This policy must be constructed by some trusted party, be it the compiler writer or some other person. The policy may specify a set of data invariants, or it may specify system calls that are not allowed to be executed. The generation of a good policy is the linchpin in the whole system. If the policy is too strict, then too many programs would be considered unsafe. If the policy is too lenient, then malicious programs will be allowed to execute.

Other researchers have extended PCC to other domains, such as authentication. In Proof-Carrying Authentication (PCA), a subject whom wishes to access some object (be it a server or some piece of data) must provide a proof that allows such access. While the purpose is different, the similarities between the PCC and PCA are straightforward to understand: In both instances, the burden of providing the proof is placed on the entity that is requiring access, instead of upon the verifier. In the case of proof-carrying code, the entity is the program trying to access the execution environment.

A problem with PCC is that even if the code is verified to be bug free, it still must be converted to a machine-readable form to be executed. Most programmers take for granted that the compiler does not introduce bugs. However, any optimizations performed can cause problems if not done carefully. Avoiding optimizations can alleviate the problem, but the runtime improvements may be necessary. There are many proposals on how to fix this issue. One approach is to certify that the compiler does exactly what it claims, which can be a complicated process. A set of compilers could be used to compile the same code and the compiled output could be compared, an approach proposed by Eide and Regehr [15] for

detecting compiler bugs in general, but finding independent implementations of compilers for PCC may be prohibitive. Another approach is to prove the correctness of the algorithms used in the compiler, but there could still be bugs in the implementations. Additionally, any proof of the implementation must be redone any time there is a change to the compiler.

## 5.1 Current Research

In 1998, Necula and Lee expanded the idea of PCC, showing that it could be used to achieve the certification as described above [28] . Using a type-safe subset of C (and adding some Java style enhancements for exceptions and heap allocation), the compiler produces annotations along with the machine code for the program. The certifier, used to verify the result of the compilation, is the same structure as the verifier from the original PCC paper. The machine code generated by the compiler is highly optimized, but the type annotations provide enough information for the verifier to deduce whether or not the machine code results in type safe code.

Necula and Lee are not the only researchers who have researched certifying code. Denney and Fischer at NASA Ames Research Center also looked at this problem [11]. Denney and Fischer claim that their research can be seen as PCC for code generators. Instead of focusing on object code generation, they focus on source code generation. Their system takes a problem specification, and produces source code and a certificate which proves that the generated code complies to a specified safety policy.

On the PCA front, Avik Chaudhuri and Deepak Garg have developed PCAL, language support for PCA systems [6]. PCAL is an extension of the Bash scripting language, where the PCA annotations are converted to regular Bash scripts. These scripts can then be used in an existing

PCA system. Contribution of this work is the language annotations, and their compiler. The annotations allow the programmer to specify what proofs they think hold at certain points of the program. The compiler then uses a theorem prover to produce the proofs statically, so they can be passed to the PCA system.

## 6 Static Analysis Techniques and Tools

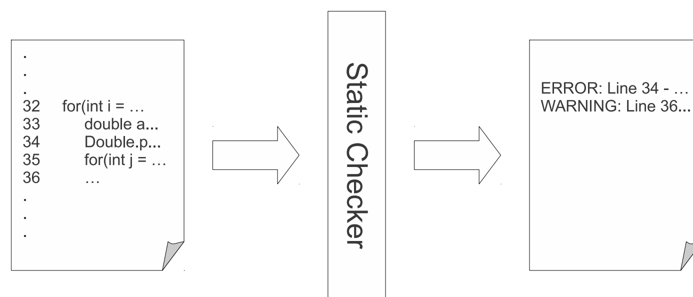


**Fig. 5.** Simple view of a static analysis tool

Static checking techniques can have their roots traced back to statically typed languages such as Fortran, C, and Java. For these programming languages, the compiler would check to make sure there were not any type violations before generating the executable. In general, that is exactly what a static analysis tool does. A static analysis tools looks for violations of some pre-defined rule, designed to cut down on the amount of debugging necessary for a programmer. Many such tools exist and they can catch errors that do not propagate at runtime [12], such as intermittent errors that only affect a small number of executions.

In this section, we will describe and compare static type checking and extended static checking. We will then discuss the state-of-the-art research in this area.

### 6.1 Static Type Checking

A type checker's main goal is to assure that all data types are correct given their context in the program. For example, the Java compiler will throw an error if a program uses the division operator on two String operands. The rules that a type checker follows stem directly from the specification of the language. The type checker for C is different from that of Java, as the syntax and semantics of the language are different. For example, programmers can "add" (use the '+' operator on) two Strings in Java, but not in C.

A typing rule that is derived from the language specification is known as *type expression*. The collection of such rules are known as a *type system*. An implementation of a type system is a *type checker*. Type checkers can be either static or dynamic. In theory, any type system can be checked statically or dynamically. In practice, however, there exist type expressions that must be checked by a dynamic type checker, such as array bounds checking. There are benefits and drawbacks to either type of checker. For example, dynamic checkers can affect the run time of a program, but also can prevent error prone code from executing (like array bounds errors which lead to buffer overflow ). A full discussion of static versus dynamic checkers is beyond the scope of this chapter. More information about dynamic type checkers can be found in section 8.

Java, C, C++ and many other languages are statically typed. This means that most of the type safety checks are performed at compile time. However, there are differences between the type checkers that stem from more than their languages' syntaxes. These differences can be described by how strongly the type system is enforced by the type checker. For example, Java may be considered more strongly typed than C and C++, as it does not allow for explicit pointer arithmetic. However, other languages can be considered more strongly typed than Java, since Java will per-

form some implicit type conversions in order to cut down on programmer effort.

Defining what constitutes a strongly typed language versus a weakly typed language is a complex task. The definitions are highly debated, but presented here are some general definitions. A language is considered *strongly typed* if all type unsafe operations are detected or prevented from occurring. Given this definition, C and C++ cannot be considered a strongly typed language, as there are some scenarios in which type unsafe operations can occur. Other definitions of strongly typed lead to C and C++ being considered strongly typed. A *weakly typed* language is any language that is not strongly typed. Languages such as Perl and Javascript are generally considered to be weakly typed languages.

A programming language is considered dynamically typed if the majority of the type safety checks are performed at run-time. Examples of dynamically typed languages are Python, Perl, and Lua. Dynamic typing does not preclude strong typing, as Python is considered a strongly typed language. Dynamic type checking opens a new door into the kinds of type checks that can be performed. These new checks can allow for code that statically typed languages would reject as being illegal to be incorporated into a program.

As previously mentioned, there are a few drawbacks to dynamic checks. Since the dynamic checks happen at run time, there is going to be a runtime penalty. Not only that, but these penalties are incurred every time the program is executed. Also, the same errors that a static type checker would quickly catch are not discovered until run time. Errors such as variable typos would result in run time exceptions. Debugging might even become more of a hassle, as errors that do get reported could have their root cause elsewhere. Of course, this problem can also exist in some static type checking techniques.

## 6.2 Extended Static Checking

Consider for a moment the problem with array bounds checking briefly mentioned in section 6.1. It was stated that bounds checking requires a dynamic type checker, because a static type checker could not determine (with the provided rules) all of the possible values an array index variable could take and whether or not it violated the array's typing rule. That does not mean that it could not be checked statically, only that the typing system rules cannot encapsulate enough information for the check.

Array bounds checking is one example of a type of problem that Extended Static Checking (ESC) can solve. There are multitudes of problems that extended static checking can solve. Extended checkers use ideas from other areas of computer science such as program verification. The main goal of these checkers are to catch program errors that normally could not be caught until run time. Sometimes, ESC can catch problems that could not even be caught at run-time, such as race conditions and deadlocks [12] [16].

Many consider ESC to have been established by Dick Sites' dissertation in 1974. The first realistic implementation of an extended checker is considered to be at COMPAQ labs in 1992, with their extended static checker for the Modula-3 language [12]. In 1997, they continued their work in ESC, developing one for the Java language [16]. In the following section, we will discuss the contributions of these, and other, static checkers.

## 6.3 Current Research

As discussed in section 6.1, there are two kinds of type checkers: static and dynamic. Each have their pros and cons. Cormac Flanagan and Kenneth Knowles have developed a novel approach to overcome these slight problems [23]. Their approach uses both static type checking and

dynamic checking in order to enjoy the benefits of both. The hybrid type checker described by Flanagan and Knowles should be applicable to many languages, as it was developed using an extension of the $\lambda$-calculus.

In hybrid type checking, the compilation phase can end in one of three states. Two states refer to the statically decidable well-typed and ill-typed phases. These programs can be decided 100% at runtime. However, some programs may not be statically determined to fall into one of these two states. In this case, the compilation phase ends in an undetermined state, and dynamic checks are inserted into the code in order to verify safety at runtime. These are referred to as *subtle* programs and can fall into two states, those where the checks always pass, and those where some checks fail. In this scheme, all well-typed programs are allowed to execute, even if they cannot be checked statically. Programs that are statically determined incur no additional runtime penalty that they would otherwise suffer in a dynamic typed language. Those that must be checked dynamically do so without additional runtime penalty over a purely dynamic typed language.

Coverity[7] is a commercial product that has been used by companies such as the Mozilla Project, Sun Microsystems and Sega. Coverity consists of several different components, one of which is what they consider a $3^{rd}$ generation static analysis tool. They extend the ideas first used in the Lint tool and the Stanford Checker (as well as other static analysis tools from the early 2000s). The result is a powerful tool that guarantees 100% path and value coverage, with low false positive rates.

Coverity combines the path simulation aspects of other tools with SAT (Boolean Satisfiability) solvers. In path simulation, the target project is converted into a graph abstraction. This is done by considering the control points in the program, which represent places in the code where behavior is determined by the current system values. Each path in this graph abstraction is tested for defects. This is a powerful technique, but

it can lead to high false positive rates because not all paths in the graph abstraction can be followed at run time. Coverity uses SAT solvers to eliminate false positives from being reported.

Due to the way the graphs are generated for the path simulation algorithms, a boolean expression can be generated for any defect found in a program. This expression is related to the control points that need to be followed, and the values that define those control points. Given the boolean expression, the SAT solver attempts to satisfy the expression using values for variables that are possible at run time (this may require adding more clauses to the boolean expression). If the expression can be satisfied, then the defect is a true defect, otherwise it is a false positive, and not reported. Even with these complex algorithms, Coverity is still able to scale to projects with millions of lines of code. This is achieved by caching results of the SAT expressions (a known SAT technique).

## 7 Information Flow Techniques

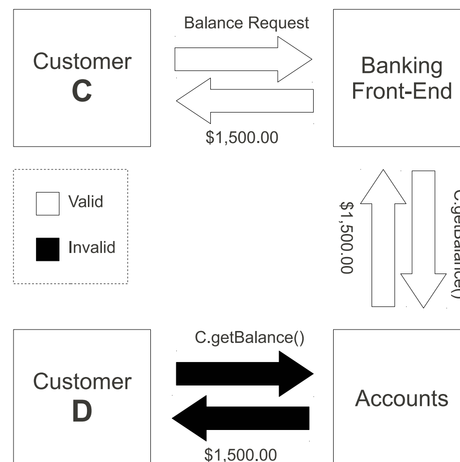


**Fig. 6.** Sample Information Flow

During the normal lifetime of an executing process, data will be read in and written out to various locations that the process has access to. For programs that do not have to worry about security, this is a non-issue. However, it is sometimes necessary to have control over what pieces of the program have access to certain pieces of data. For example, it is usually a good idea to not let data encryption keys be sent through print functions (or at least limit the cases when this is permissible). Being able to keep track of where data is being used in a program is a useful tool for these situations.

Information flow tracking has been studied since the 1970s. The idea is that the program is analyzed to determine whether or not illegal data flows occur. Of course, the notion of an illegal data flow is similar to the "bad" program operations described in section 5. Someone has to define what is an illegal data flow for a particular program. These policies are defined in a specification language, which are often based off of access control techniques.

In this section, we will briefly discuss general information flow policies. Then, we will describe the differences between static and dynamic information flow techniques. Finally, we will discuss current research in the area, which includes language specific tools or language modifications designed to support information flow techniques.

### 7.1 Introduction to Information Flow

There are two types of information flow: *explicit* and *implicit*. Explicit information flow occurs when there is direct transfer of information between two entities. For example, in the assignment statement `x = y`, information explicitly flows from `y` to `x`. Explicit flows generally occur at assignment statement. Implicit flows occur when the value of one variable (say `x`) can be inferred by examining the value of another (`y`). Consider the statement `if(x < 1) then y = 7`. If after passing that control se-

quence, the value of `y` is 7, then we know a restriction on the value of `x`. There are also *direct* and *indirect* information flows. The above explicit example is also an example of a direct information flow, since the information is directly gained via the assignment. If there is a layer of indirection [e.g. `z = x; y = z`], then it is considered an indirect information flow (since information of `x` is found in `y` only through `z`).

### 7.2 Static vs. Dynamic Information Flow Techniques

By definition, *static* information flow techniques are performed without executing the program, while *dynamic* information flow techniques are performed while the program is executing. As one would imagine, the pro's and con's of static versus dynamic are similar throughout this chapter. Static does not affect the program's execution time, but is not as precise as dynamic. Static information flow can prove that there are not insecure information flow, but may reject some programs that do not have invalid information flow during execution. Dynamic is able to be more precise (since it catches errors right before they propagate), but gains that ability at the cost of execution time. Dynamic information flows can look for attack signatures to catch related information flow attacks and resist evasion attempts.

### 7.3 Current Research

A.C. Myers et al. have extended the Java language to include information flow annotations. In JFlow [26], data values are annotated with labels. These labels are a set of security policies, which may come from different principals (an abstraction of a user or set of users). One of the major contributions of this work is the decentralized label model, which allows for the owners to be mutually distrusting, and the policies to still be held. The policies in the labels can be checked statically, and show that

information is not leaked in the program. It can also perform run-time checks if the static checking proves to be too restrictive.

JFlow is a source to source translator that also has a static checker. Most annotations are removed during this process. Most notably, since they are statically checked, all static labels are removed without affecting the run-time of the java program. Some expressions get translated to equivalent Java code that accomplishes the original goals. There are a few built in types for the run-time systems, which allow for the dynamic checks of labels and principles as needed.

Another contribution of this work is the idea of label polymorphism. Classes in Java are written to be generic as possible, to allow reuse in a number of applications. Consider the Java Vector class, a class that can store an arbitrary number of object. This class in parameterized in the Java standard library, which allows the user to generically store a specific type of object in the vector. This allows the vector class to store a specific, but arbitrary type of data. The same concept applies for labels. The vector class is parameterized for a generic label, which does not have to be specified until it is actually being used (e.g. object instantiation). Without this ability, a different Vector class would have to be written for each label type. The same idea applies to method parameters, as they can use generic labels so only one method has to be used.

A.C. Myers has continued to work in the area of language-based information-flow. In a 2005 article on the subject [33], A.C. Myers and A. Sabelfeld discussed the current trends in the area. They discuss extensively on how new research has handled issues in the areas concurrency, covert channels, language expressiveness, and security policies. The important contribution of this work is their discussion of open challenges in the area of information-flow. One such issues has been touched on briefly already in this chapter, certifying compilation (see section 5.1). Current information flow research assumes the compiler to be in the trusted computing

base, but it is desirable to remove this necessity. Next, the idea of system-wide security requires information-flow tracking to be integrated with the rest of the system security mechanisms to provide a secured computing environment. Finally, there is a discussion of the practical issues involved with the language-based modifications for not only information-flow, but all security mechanisms.

# 8 Rule-checking, Verification and Runtime Support

Runtime systems have been around since the 1950's, starting with the Lisp interpreter. Software verification has been a topic of concern for a similar time frame. Research in these areas have been considerable since that time. High profile languages such as Java are interpreted just as Lisp has been. Other research projects have added run-time systems to languages such as C, to aid in the security and usability of these languages. Software verification has come a long way in that time frame as well. In this section, we will discuss some recent advances in these technologies.

## 8.1 Current Research

Formal methods have a rich history, but usage in security has not been widespread. One place where they are being used is at Microsoft. In 2004, Thomas Ball et al. published a paper on the SLAM project [3]. SLAM is an analysis engine that checks to make sure a program adheres to the interfaces provided by an external library. This project is the basis of Microsoft's Static Driver Verifier (SDV), which is now part of the driver signing procedure for Windows.

Performance of the original SLAM project was fine, but they had a non-negligible amount of false positives in their checks. At just over 25%, it

generally meant that some programmer would have to check each alarm to see if the alert was correct. Thomas Ball et al. decided to try to fix this problem by fully re-implementing the entire system, improving the areas that they found were causing the most errors. They were able to get the false positive rate to a manageable 4% in their new SLAM 2 system [2].

The Microsoft Singularity Project [37] started in 2004 as an attempt to create a secure, dependable operating system. While it is an OS, work on the Singularity project is also influencing the area of language research as well, defining their own *C#* dialect called *Sing#*. In Singularity, each application is executed in isolation, a model known as Software-Isolation Processes (SIP). Each SIP is a closed object and code space, meaning that objects cannot be shared between SIPs, and code cannot load or generate code after it begins execution. In order for such a system to work, a communication mechanism needs to be established.

Singularity provides bi-directional channels for processes to communicate over. Each channel has a contract, which defines what messages are transmitted over the channel and in what order to expect them. These contracts can be statically enforced, and can be interpreted by the runtime garbage collector to determine when such static enforcement is unable to be performed. The channel contracts are a language feature which allows for the communication to be checked, and their experience with the implementation has been a positive one [18].

CoreDet [4] is a compiler and runtime system for deterministic execution of multithreaded programs. This is accomplished without language modifications or hardware support. Bugs that result from Non-deterministic behavior is very difficult to reason about. Executing code in a deterministic way allows for programmer to replicate bugs easily, in order to solve such issues. This work is an extension of the work from the DMP [13] project, implementing the algorithms proposed in a deployable runtime system.

There are several modes which CoreDet can operate in. The most basic mode is a serial mode, where every thread is executed serially. This is a simple way to solve the non-determinism problem, but is also the slowest. DMP-O is a mode where ownership of data is tracked on a per thread basis. As long as threads are accessing data that they "own," then the threads are allowed to execute in parallel. When data is accessed that is not "owned" by a thread, all threads execute in serial until the access is finished. DMP-B executes all threads in parallel, on a segment by segment basis. Each thread as a private buffer of data that is used in each segment. At the end of each parallel segment, the threads enter a serial commit phase, where values from each private buffer is committed to the global memory space. DMP-PB combines DMP-O and DMP-B to increase the scalability of the system.

Over all of the modes, DMP-PB scales the best as the number of threads increases. When compared to non-deterministic execution, there is performance loss for all of the modes in CoreDet. This is to be expected, since all modes have at least one sequence where code is executed in a sequential manner. However, as more CPU cores are used, the difference between CoreDet and non-deterministic execution closes. These results rely on optimal configuration settings for each, but the authors claim that the benefits of deterministic execution is worth the degradation in performance.

# 9 Language Modifications for Increased Safety and Security

As discussed in sections 6 and 4, we discussed how some languages are not as "safe" as other languages. Programs written in C are susceptible to attacks (such as the buffer-overflow attack) that the Java language is not. Many legacy programs are written in these "unsafe" languages,

and are vulnerable to such attacks. Furthermore, new programs continue to be written in these languages. In this section, we will survey projects that have modified existing languages in an attempt to make them more safe.

## 9.1 Current Research

One of the most common complaints about Java is its performance. Because of interpretation overhead, as well as the run time safety checks, Java programs have trouble keeping up with the performance of C or C++ programs. One way to overcome this deficiency is to incorporate native-code into the program, using the Java Native Interface (JNI). Unfortunately, this bypasses the security model in Java. Robusta [36] is a modification of the Java virtual machine, which isolates native code from affecting the rest of the program.

Robusta uses the Native Client (NaCl) project [43], a software-based fault isolation system from Google. NaCl is modified to comply with the standard Java programming model, such as allowing dynamic linking and loading of classes. Robusta also inserts safety checks before an after each JNI call. These calls ensure that the native code does not bypass any of the JVM's security mechanisms. For example, when native code calls a Java method, Robusta checks to make sure that all parameters are of the correct type. It is able to achieve this through the use of *trampoline* and *springboard* functions, which are the only functions that are allowed to cross the boundary of the isolated environment.

All tests performed show decent performance for a series of JNI benchmarks. Most overheads were less than 10%, although there were two benchmarks that had substantial overheads. The authors of Robusta state that this is likely due to their high number of context switches, which were orders of magnitude higher than the other benchmarks. Comparing to other JNI safety systems, the Robusta overheads are promising.

Many programmers like the control that the C language provides. The precise control over memory management and data structures gives the programmer the ability to manage the execution of the program as they see fit. Other languages lack the control that C provides, but lack the problems that C can have, such as buffer overflows and out-of-bounds pointers. Cyclone [20] tries to gain the safety of languages such as Java and Lisp, without losing all of the control that C has.

All of the changes to the C language in Cyclone are to handle potential safety violations. Most of the changes deal with how pointers are handled. For example, Pointer arithmetic is restricted, in an attempt to prevent certain buffer overflow attacks. Only certain types of pointers can have arithmetic performed on them. These pointers contain bounds information, which is used to check the arithmetic performed, to ensure the new pointer is still within the bounds of the allocated range.

One of the most drastic change is how memory is managed is handled in Cyclone. Since `free` can cause dangling pointers, they decided to make `free` a no-op. They provide two new ways to reclaim unused memory. First, they provide an optional garbage collector. Second, to provide finer-grained control of memory, they provide a new feature called a *growable region*. Regions are code blocks where all data allocated in the region is implicitly deallocated at the end of the region block. These regions have handle variables, which can be passed to other functions as necessary. Static analysis can then determine if the variables that are declared within the region are used in a way that could cause safety violations, such as storing pointers into the region in a global variable.

Many other restrictions exist, but full discussion is out of the scope of this section. During testing, Cyclone was able to find errors in a few of the benchmarks. There was overhead caused by Cyclone, most of which was caused by the bounds checking performed. Most I/O bounded applications experienced nominal overheads, but computationally intensive

programs had considerable overhead. The authors claim that some programs took six times as long to execute. The authors were able to achieve they goal to bring safety guarantees to a low level language such as C.

CCured [29] has a similar motivation as Cyclone. Both languages attempt to bring safety to the C language without major syntax changes. In CCured's case, George Necula et al. attempt to make C type safe. CCured is a two part system. First, there is a static analyzer which verifies type safety at compile time. Second, run-time checks are performed to check those portions of the program that cannot be verified statically.

The major contribution of the CCured work is the creation of three separate pointer types. First, there is a special pointer type for pointers accessing arrays. These pointers are bounds checked when dereferenced, to make sure the reference stays within the bounds of the memory region. Second, there are safe pointers. These pointers are always either null, or valid. They are not allowed to have pointer arithmetic performed on them. Finally, there are dynamic pointers. These pointers encompass all other forms of pointers, but they still have bounds checking performed on them upon dereference. CCured deals with dynamic memory similar to how Cyclone does, by ignoring explicit frees. Instead, CCured relies on a garbage collector to reclaim unused memory. They claim that it is possible to use a precise garbage collector, but leave that to future work

Another of the major contributions of CCured is its type inference engine. One of their main goals was to be able to take legacy C code, and allow CCured to work directly with it without having to annotate the code. They simply could change all pointers to be of the dynamic type, but there are benefits to the safe and sequence pointers (such as skipping bounds checks for safe pointers, or saving some space for sequence pointers). The inference system finds the minimum number of dynamic pointers, then makes the rest of the pointers sequential if any pointer

arithmetic is performed on them. The remainder are defined as safe. The inference algorithm is linear in the size of the program.

The performance of CCured is not great. The performance is not as bad as CCured without the type inference engine, but the performance ranged from 1.03 times to 2.44 times longer execution time. They were able to fix several bugs in the SPECINT95 benchmark suite. Their work performs much better than the Purify [19] tool, which instruments binaries to detect memory errors.

## 10 Conclusions and Future Work

We have only scratched the surface of language and software based protection mechanisms in this chapter. Language choice when writing software can have a major affect on the vulnerabilities present in the final program. The environment the program is executed in has a similar affect. The techniques described can help protect software in spite of the flaws in the default language tools and environment. Even with the security gains we have discussed, there are still many issues that need to be solved.

Protecting software from piracy can be accomplished using watermarking or obfuscation. However, current techniques use cryptographic or runtime protection mechanisms. Some forms of piracy protection have been met with large amounts of opposition from the end users, mainly due to the restrictions placed on the usage of the software. Future work will look at how the mechanisms discussed here can be used to protect software from piracy without harming the experience of the end user.

Another trend in the current technological environment is the move towards mobile computing devices. Most mobile phone environments, such as the Android OS, have their own protection mechanisms in place in order to mitigate the amount of damage to the system in the presence of

a malicious application. Still, devices such as the iPhone have had published vulnerabilities, even if they only affect user-modified phones. Even the default Android OS has had its fair share of malicious programs as well. Future work in software protection will have to consider protecting mobile devices.

Stack-based buffer overflow exploits are still relatively common. They are no longer the most common attack on a software system. The drive towards cloud computing has converted many software systems into internet applications. Cross-site scripting (XSS) attacks, as well as SQL injections have become the attack of choice for these applications. In an SQL injection attack, a malicious user is able to execute SQL queries in order to modify the underlying database for the application. While a large number of SQL injection attacks can easily be defended by properly sanitizing user data before usage in SQL statements, others can be fairly difficult to defend against. XSS attacks allow a malicious entity to inject code into another, generally trusted website. The injected code is then run on the client machine. Protecting internet applications against these attacks is an important area of future work.

## Exercises

1. Discuss the similarities and differences between watermarking protocols discussed in section 2, and a cryptographic protocol. Is watermarking a form of steganography?
2. In section 3, we discussed several uses of code obfuscators. A common technique when writing code in J2ME (Java 2 Micro Edition, commonly used in mobile phone development) is to perform an obfuscation as the last step. Other than an increase in security, what other benefit can be pulled from performing such obfuscations.

   **Hint:** You might want to compile a sample program before and after a simple abstraction transform, to compare class files.

3. Consider dynamically typed languages such as `Python`. Are these languages easier to obfuscate than their strongly typed counterparts? Why or why not?
4. Write a `C` program to perform a stack-based buffer overflow attack as described in section 4. Does your compiler automatically inject stack protection? If not, can you force it to do so?
5. How would you determine the layout of the heap to perform a heap-based buffer overflow attack? Are there programming languages that are naturally resistant to heap based attacks?
6. Brainstorm some uses for the information flow tracking discussed in section 7.
7. Many of the `C` based tools and techniques discussed here require a modification of the language itself. Does this mean that the language is inherently unsafe to use in its general form? Why is `C` so prominently used over "safer" languages such as `Java`?
8. Throughout this chapter, we have discussed the differences between static and dynamic protection mechanisms. In the field of cyber-physical systems, the trade-offs between execution time and precision are important. For each of the following examples, discuss which type of protection mechanism (static or dynamic) is best suited:
   - Spy Satellite
   - Air traffic controller system
   - Smartphone

## References


1. Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20, New York, NY, USA, 2007. ACM.

2. Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2: Static driver verification with under 4 percent false alarms, 2010.
3. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2004.
4. Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 53–64, New York, NY, USA, 2010. ACM.
5. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.
6. Avik Chaudhuri and Deepak Garg. Pcal: Language support for proof-carrying authorization systems. In Michael Backes and Peng Ning, editors, *Computer Security ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 184–199. Springer Berlin / Heidelberg, 2009.
7. Ben Chelf and Andy Chou. The next generation of static analysis: Boolean satisfiability and path simulation - a perfect match. White paper, Coverity inc.
8. Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.*

Addison-Wesley Professional, 2009.

9. Crispin Cowan, Dylan McNamee, Andrew Black, Calton Pu, Jonathan Walpole, Charles Krasic, Perry Wagle, Qian Zhang, and Renauld Marlet. A toolkit for specializing production operating system code. Technical report, 1997.
10. Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.
11. Ewen Denney and Bernd Fischer. Certifiable program generation. *GPCE 2005, LNCS*, 3676:17–28, 2005.
12. David L. Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical report, COMPAQ, 1998.
13. Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.
14. Jim Duffey. Software piracy rate up 2% in 2009, study finds. *Network World*, May 2010. http://www.networkworld.com/news/2010/051110-software-piracy.html.
15. Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 255–264, New York, NY, USA, 2008. ACM.
16. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for

java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

17. Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 15–26, New York, NY, USA, 2008. ACM.
18. Manuel Fhndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the EuroSys 2006 Conference*, pages 177–190, New York, NY, USA, 2006. Association for Computing Machinery, Inc.
19. Reed Hastings and Bob Joyce. Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, Berkeley, CA, USA, 1992. USENIX Association.
20. Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.
21. Steve Jobs. Thoughts on music, February 2007. http://www.apple.com/hotnews/thoughtsonmusic/.
22. Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. *SIGARCH Comput. Archit. News*, 34:207–218, October 2006.
23. Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):1–34, 2010.
24. Matias Madou, Ludo Van Put, and Koen De Bosschere. Loco: an interactive code (de)obfuscation tool. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and*

*semantics-based program manipulation*, pages 140–144, New York, NY, USA, 2006. ACM.

25. Matias Madou, Ludo Van Put, and Koen De Bosschere. Understanding obfuscated code. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 268–274, Washington, DC, USA, 2006. IEEE Computer Society.
26. Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.
27. George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.
28. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.
29. George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. *SIGPLAN Not.*, 37:128–139, January 2002.
30. Landon Curt Noll, Simon Cooper, Peter Seebach, and Leonid A. Broukhis. The international obfuscated c code contest. http://www.ioccc.org/.
31. Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.
32. Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2004.
33. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, jan. 2003.
34. Securom ™. http://www2.securom.com/.

35. Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

36. Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: taming the native beast of the jvm. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 201–211, New York, NY, USA, 2010. ACM.

37. Singularity - Microsoft Research. http://research.microsoft.com/en-us/projects/singularity.

38. Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702.

39. Stackshield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield.

40. Ramarathnam Venkatesan, Vijay V. Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop on Information Hiding*, IHW '01, pages 157–168, London, UK, 2001. Springer-Verlag.

41. Wikipedia. Content scramble system — wikipedia, the free encyclopedia, 2011.

42. Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. Mimimorphism: a new approach to binary code obfuscation. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 536–546, New York, NY, USA, 2010. ACM.

43. Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas

Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53:91–99, January 2010.

44. Changjiang Zhang, Jianmin Wang, Clark Thomborson, Chaokun Wang, and Christian Collberg. A semi-dynamic multiple watermarking scheme for java applications. In *DRM '09: Proceedings of the nineth ACM workshop on Digital rights management*, pages 59–72, New York, NY, USA, 2009. ACM.

# Index

boolean satisfiability, 27
buffer overflow, 15, 24, 36
– heap-based, 19
– stack-based, 17

compiler, 20, 33
– certifying, 22
cross-site scripting, 39

digital rights management, 2, 6, 9
DRM, *see* digital rights management

formal methods, 32

information flow, 29

malware, 9, 13

obfuscation, 7

PCC, *see* Proof Carrying Code
piracy, 2
Proof Carrying Code, 20

runtime, 32

SQL injection, 39
static analysis, 23
– type checking, 24

watermark, 3