Securing Software Applications Using Dynamic Dataflow Analysis

Steve Cook

Southwest Research Institute
scook@swri.org
(210) 522-6322

OWASP
June 16, 2010

The OWASP Foundation
http://www.owasp.org

Outline

- Introduction and Overview
- How DDFA Works
- Illustrative Example Scenarios
- Efficiency of DDFA
- Wrap Up

What is DDFA ?

- DDFA is an extensible compiler-based system that automatically instruments input C programs to enforce a user-specified security policy

- Approach uses a complementary combination of static and dynamic data flow analysis along with the policy to produce secure programs with low runtime overhead

DDFA Development Team

- University of Texas at Austin, Computer Science
  - Fundamental research on Dynamic Dataflow Analysis
- Southwest Research Institute
  - Applied research and tech transfer

Why is DDFA Needed ?

- Widespread use of untrusted COTS / Open Source software
- Large legacy code bases
- Programs not designed with security in mind
- Difficult and costly to find software developers well-versed in application security

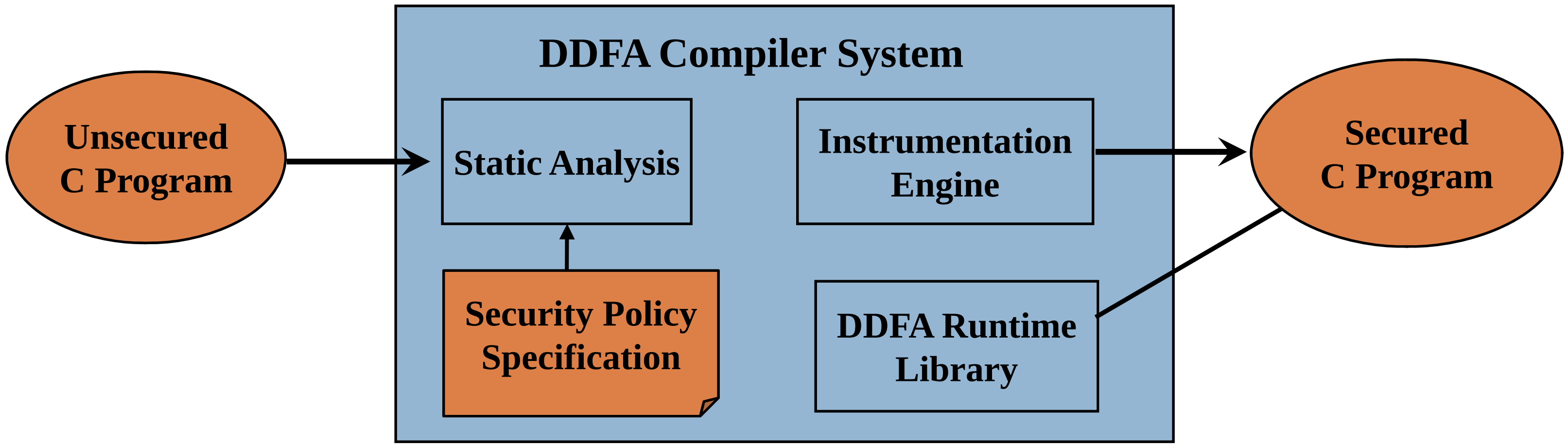
Research Goals

- Minimize the impact to software development
  - Easy to use and deploy
  - Provide separation of concerns
- Keep program runtime and size overhead as low as possible
- Support multi-level security
  - Not just one binary state (e.g. bad, good)
- Provide extensibility for future threats

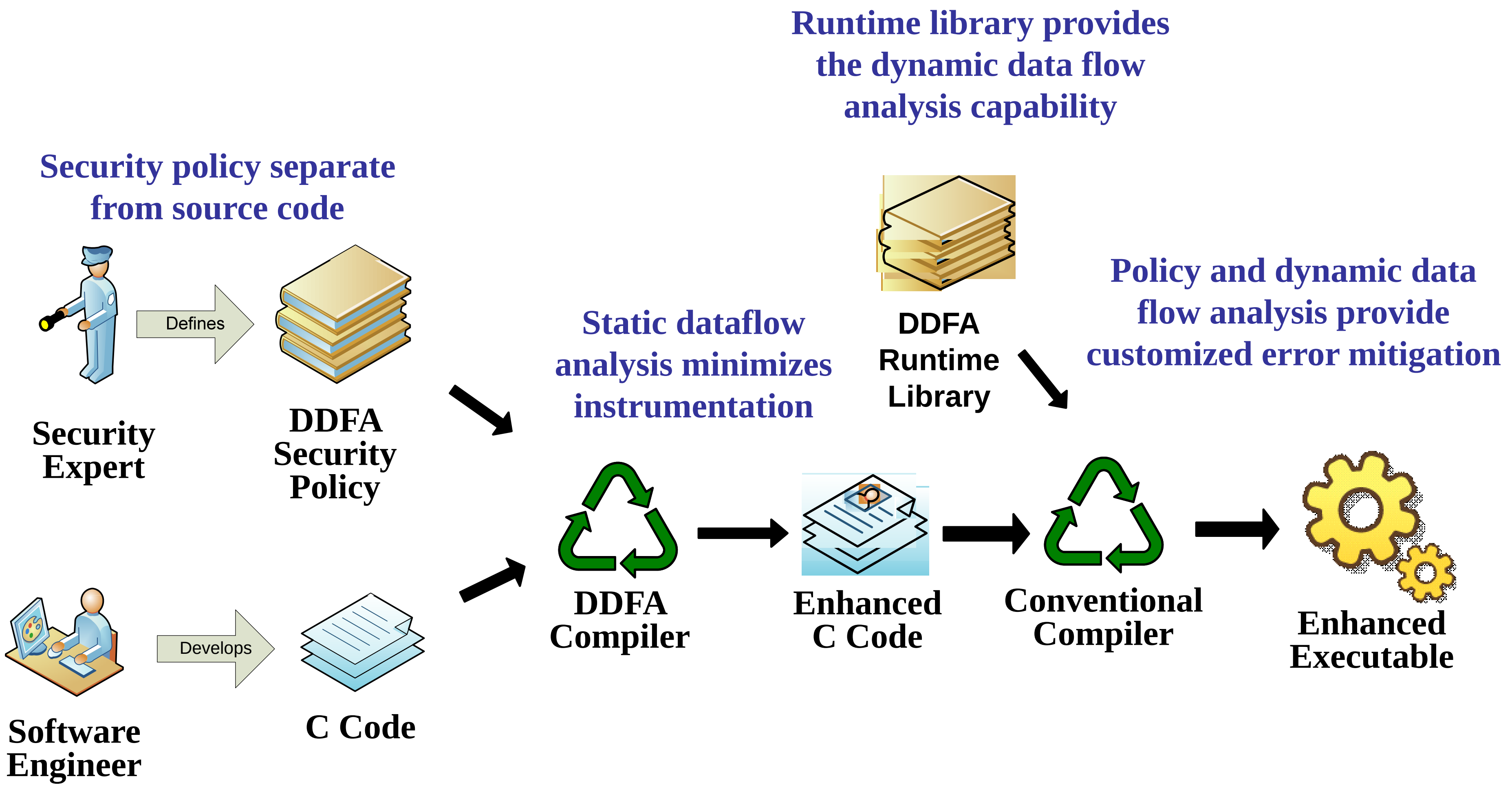
State of the Art

- Manual code inspection that support best practices
- Many automated approaches focus only on memory safety
  - Less important as memory-safe languages such as Java become more popular
- Static Analysis Tools (e.g. Coverity)
  - Statically detect bugs and vulnerabilities
  - Admits both false positives and false negatives
  - Only detects bugs, does not fix them
- Taint Tracking approaches
  - High runtime overhead (82% - 7.9×)
  - Not general enough for multi-level security

Architecture of DDFA System



Development with DDFA


Runtime library provides the dynamic data flow analysis capability
Security policy separate from source code
Defines
Security Expert
DDFA Security Policy
Static dataflow analysis minimizes instrumentation
DDFA Runtime Library
Policy and dynamic data flow analysis provide customized error mitigation
Develops
Software Engineer
C Code
DDFA Compiler
Enhanced C Code
Conventional Compiler
Enhanced Executable

Primary Benefits of DDFA

- Application dataflow is tracked at compile and run time
  - Very low runtime overhead (many cases < 1%)
    - Leverages semantic information from policy
  - Configurable error mitigation at run time (e.g. fight through)
- Policy is separate from the source code
  - Removes security concerns when developing new applications
    - Including 3rd party and open-source development
  - Can secure existing legacy applications
  - Requires one additional step in an automated build process
  - Defined once and used many times
  - Policy can change and be re-applied as threats evolve

Generality of the DDFA Approach

- Traditional Tainted Data Attacks
  - Format String Attacks
  - SQL Injection
  - Command Injection
  - Cross-Site Scripting
- Other Security Problems
  - File Disclosure Vulnerabilities
  - Labeled Security Enforcement
  - Role-Based Access Control, Mandatory Access Control
  - Accountable Information Flow

Outline

- Introduction and Overview
- **How DDFA Works**
- **Illustrative Example Scenarios**
- **Efficiency of DDFA**
- **Wrap Up**

Format String Vulnerability (FSV)

```
int sock;
char buf[100];
sock = socket(AF_INET, SOCK_STREAM, 0);

recv(sock, buf, 100, 0);
```

- String containing malicious formatting directives introduced into program from outside the system

```
printf(buf);
```

- Formatted output family of functions can cause target computer to execute arbitrary commands
  - e.g. printf(), sprintf()

Property Definition for FSV

- Security policy begins by defining one or more properties

```
property Taint  :  { Tainted, { Untainted } }
                   initially Untainted
```

- Each property represents a lattice
  - Lattices intrinsic to data flow analysis
  - Lattice nodes represent possible flow values
  - Flow values are meta-data attached to program objects

<u>Lattice with Two Nodes</u>

$$\begin{tikzcd}\textbf{Untainted} \arrow[d] \\ \textbf{Tainted}\end{tikzcd}$$

Annotations for Library/System Calls (Focus is on Three Areas)

- Introduction
  - Associates property values (or metadata) to memory objects as they are introduced into a program
- Propagation
  - Tracks the flow of memory objects and their property values throughout the program
- Violation
  - Identifies if a violation occurs at runtime based on the memory objects' property values, which static analysis alone is not able to do

Policy - Annotating the Library Procedures (FSV)

Original Source Code

Introduction

```
int sock;
char buf[100];
sock = socket(AF_INET, SOCK_STREAM, 0);

recv(sock, buf, 100, 0);
```

Propagation

```
buf2 = strdup(buf);
```

Policy Violation

```
printf(buf2);
```

Annotated Procedures

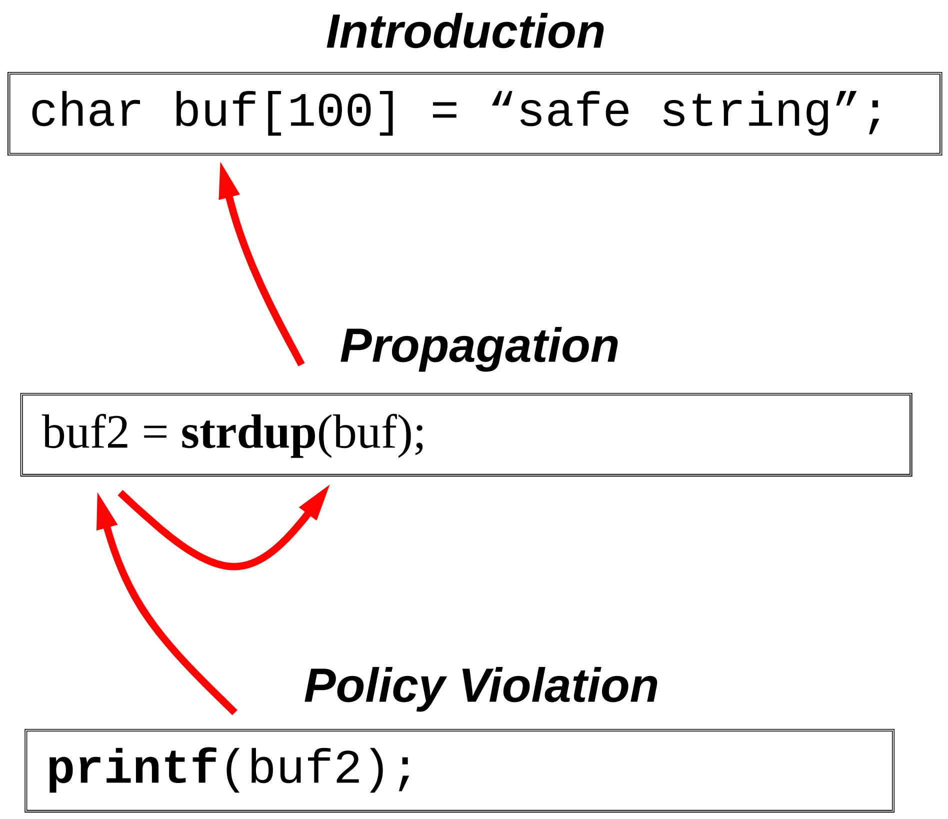
```
procedure recv(s, buf, len, flags)  {
   on_entry  { buf → buffer }
   analyze Taint  { buffer ← Tainted }
}
```

```
procedure strdup(s)  {
   on_entry  { s → string }
   on_exit  { return → string_copy
   analyze Taint  { string_copy ← string }
}
```

```
procedure printf(format, args)  {
   on_entry  { format → format_string }
   error if ( Taint: format_string could-be Tainted ) {
       error_handler = fsv_error()
       certify = fsv_check(format, args)
   }
}
```

Static Data Flow Analysis (Works Backwards)

In this case, data flow analysis proves that dynamic data flow analysis is not necessary. **No instrumentation is needed.**



Introduction

```
char buf[100] = “safe string”;
```

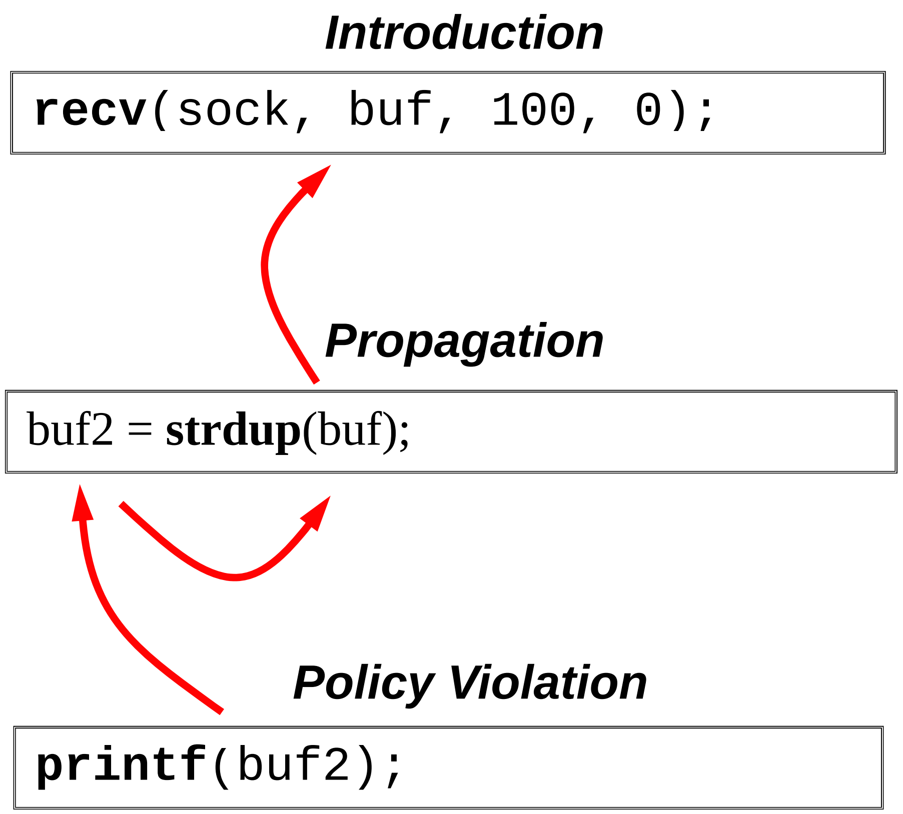
Propagation

```
buf2 = strdup(buf);
```

Policy Violation

```
printf(buf2);
```

In this case, data flow analysis determines that dynamic data flow analysis is necessary. **Source code must be instrumented.**



Introduction

```
recv(sock, buf, 100, 0);
```

Propagation

```
buf2 = strdup(buf);
```

Policy Violation

```
printf(buf2);
```

Instrumentation for Dynamic Data Flow Analysis

Program is augmented with calls to DDFA library to perform dynamic data flow analysis.

Introduction

```
recv(sock, buf, 100, 0);
ddfa_insert(LTAINT, buf, strlen(buf), LTAINT_TAINTED);
```

"buf" takes on flow value *Tainted*, since comes from outside system

Propagation

```
buf2 = strdup(buf);
ddfa_copy_flowval(LTAINT, buf2, buf, strlen(buf2));
```

Copies flow value from "buf" to "buf2"

Policy Violation

```
if ( (ddfa_check_flowval(LTAINT, buf2, LTAINT_TAINTED)) &&
(! fsv_check(buf2)) )
{  fsv_error();  }
else
{  printf(buf2);  }
```

For this flow path, "buf2" will be *Tainted*, but policy allows "Fight Through" capability using fsv_check() so error handler called only as last resort

Outline

Example 1 - Format String Vulnerability

Introduction

Hacker introduces mal-formed printf() format string via web

DDFA marks data entering from the web as "Tainted"

Propagation

```
int sock;
char buf[100];
sock = socket(AF_INET, …);

recv(sock, buf, 100, 0);
```

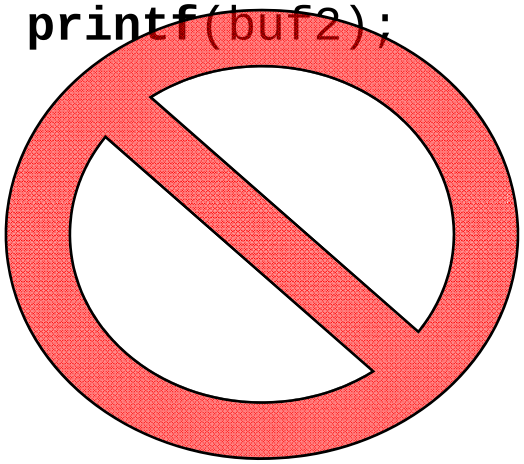
$\downarrow$

```
buf2 = strdup(buf);
```

DDFA tracks the flow of this "Tainted" data throughout the execution

Violation

```
printf(buf2);
```



Tainted string arrives at printf() statement

DDFA flags a runtime violation, preventing the vulnerability from being exploited by the hacker

Example 1 - Format String Vulnerability

- What you can’t see
  - Static analysis dramatically prunes the amount of dynamic data flow tracking
  - Pruning is enabled by the annotation-based compilation system
  - This pruning requires precise pointer analysis

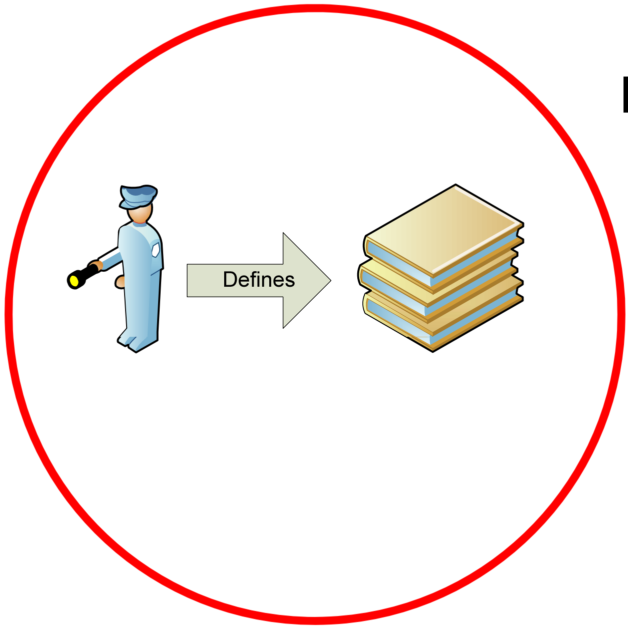
Pointer Analysis

- Pointer analysis: Tells the compiler which regions in memory pointers point to

- Pointer analysis is fundamental to all static analyses, not just DDFA

- A difficult problem:
  - Severe tradeoff between precision and scalability
  - DDFA requires a fairly precise degree of precision (flow-sensitivity)

Alternative Scenario for Example 1



- Security expert wants to fight through attacks rather than simply detect attacks
  - Takes existing security policy
  - Modifies policy to include call to new C code to sanitize Tainted data

```
if (procedure printf(fmt, args)
{
    on_entry { fmt --> format_string }
    error if (Taint: format_string could-be Tainted)
        printf(sanitize(fmt), args);
}
```

Example 2 – File Disclosure Vulnerability

Introduction

Hacker sends mal-formed "finger" packet to retrieve contents of a password file

DDFA marks Trust of finger packet as "Remote"

Propagation

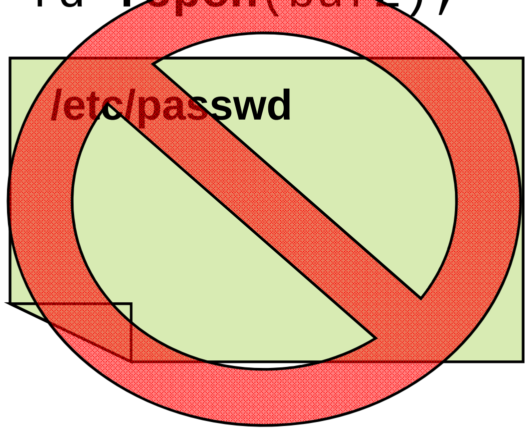
```
int sock;
char buf[100];
sock = socket(AF_INET, …);

recv(sock, buf, 100, 0);
```

```
buf2 = strdup(buf);
```

DDFA tracks the flow of this finger packet throughout the code

Violation

```
fd=fopen(buf2);
```



Data tagged as "File" originating from a "Remote" source arrives at a socket write()

DDFA prevents vulnerability from being exploited

Example 2 – File Disclosure Example

- What is interesting in this example
  - Must track both Trustedness of data and Origin of data
  - Two properties instead of one are defined in policy
  - DDFA is able to enforce multiple properties simultaneously

Example 3 – Role Based Access Control

Introduction

Beetle Bailey logs on to Missile system to perform safety checks

DDFA registers him to the system as "grunt" level

Propagation

```
ac_level = authenticate();
```

…

```
safety_check();
```

DDFA tracks the flow of all Beetle's activities throughout the missile system application

Violation

```
launch();
```

Beetle accidentally attempts to invoke launch()

DDFA flags a runtime violation, preventing missile from being launched

Example 3 – Role Based Access Control

- What's interesting in this example?
  - New functionality added to the system after development
- Separation of concerns
  - Software is difficult to build and maintain
  - Software developer should focus on core functionality
  - Security expert focuses on security (site-specific security)
  - Compiler ensures that security code is correctly and thoroughly applied
  - Separation of concerns simplifies each task

Outline

- Introduction and Overview
- How DDFA Works
- Illustrative Example Scenarios
- **Efficiency of DDFA**
- **Wrap Up**

Efficiency for Server Applications (FSV)

Program	Original	DDFA	Overhead
pfinger	3.07s	3.19s	3.78%
muh	11.23ms	11.23ms	< 0.01%
wu-ftp	2.745MB/s	2.742MB/s	0.10%
bind	3.58ms	3.57ms	< 0.01%
apache	6.048MB/s	6.062MB/s	< 0.01%
Average Increase			0.65%

Compare with 80% - 35× overhead for previous state of the art in software-based approaches

Efficiency for Compute Bound Applications (FSV)

Program	Overhead
gzip	51.35%
vpr	0.44%
mcf	< 0.01%
crafty	0.25%
Average Increase	12.93%

Synthetic vulnerabilities were inserted into programs

Original programs contained no FS vulnerabilities; true overhead is 0%

Static Code Overhead (FSV)

Program	Original	DDFA	Overhead
pfinger	49,655	49,655	0%
muh	59,880	60,488	1.01%
wu-ftp	205,487	207,997	1.22%
bind	215,669	219,765	1.90%
apache	552,114	554,514	0.43%
Average Increase			0.91%

(Size in bytes)

Table excludes other programs where static analysis proves that no instrumentation is needed

Outline

Other Potential Uses of DDFA

- Fault Tolerance Computing
- Privacy
- Testing

Future Plans

- Retarget for popular open-source compiler infrastructure, LLVM (Low-Level Virtual Machine)
  - Supports C, C++, Java on the way
- Support other languages, and possibly byte-code or binary as input

Questions