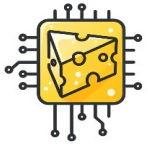
Compiler Internals

For Security Engineers



Marion Marschalek
pinkflawd@gmail.com

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and

What security relevant aspects do compilers provide?

- Mitigations
- Optimizations
- Obfuscation
- Analysis
- Instrumentation
- Intermediate representations to no end

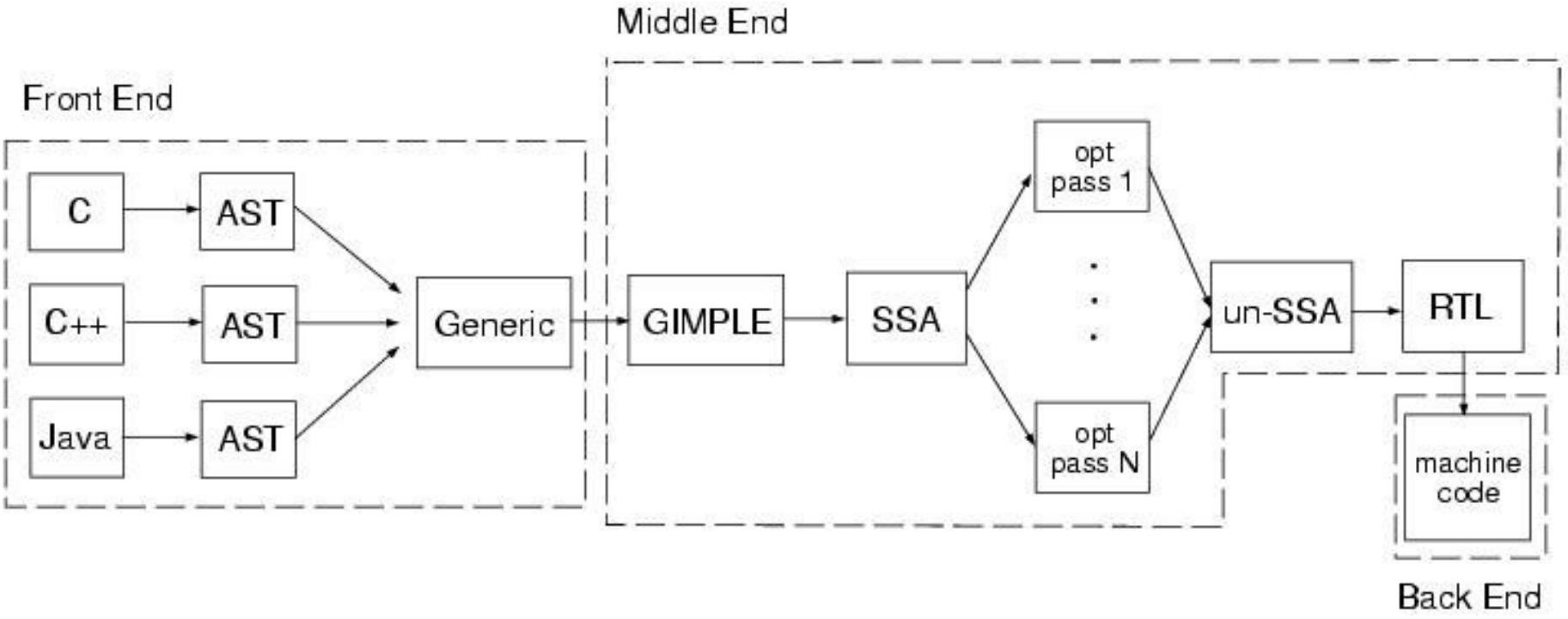
How are ELFs made?

`Source → Compiler → Assembler → Linker → Loader`

- Compiler: GCC, LLVM/Clang, Intel Compiler, VisualStudio Compiler, and so many more!
- Assembler: Converts assembly language (text) to machine readable code, produces object files
- Linker: Combines one or more objects to create executable, resolves external references and symbols
- Loader: Loads executable and maps it to process memory space, then executes it

The GCC Compiler

A 1 Mio. Foot View!



The Debug Output

… is worth gold, and looks a bit like a “Matrix” screensaver when you scroll down fast

-fdump-passes

-fdump-tree-all, -fdump-ipa-all, -fdump-rtl-all

-fdump-tree-cfg-all

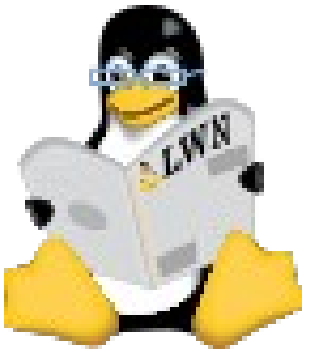
-fdump-rtl-MYAWESOMEPLUGIN

GCC Plugins

Since GCC 4.5 we can plug passes into the compilation process!

Benefits of plugins vs. modifying GCC itself?

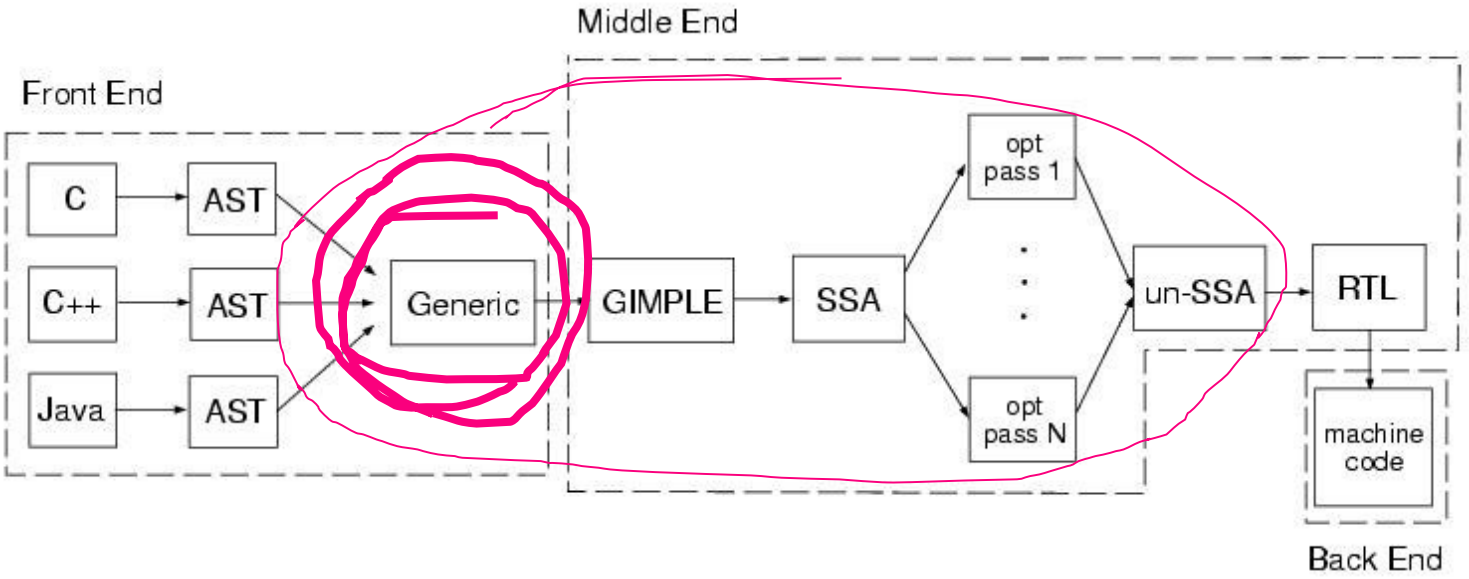
- Plugins are shared objects, loaded by GCC as dedicated passes
- Maintained by pass manager
- Dependent on compiler version
- GCC plugin API defined in tree-pass.h



https://lwn.net/Articles/457543/

GCC Representations and Data Structures

GENERIC or the mystic TREE

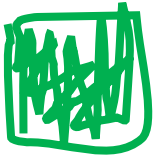


```
@3215   function_decl    name: @3217     type: @2980     scpe: @176
                         srcp: helloworld.c:3            link: extern
                         body: @3218

 ..........
@3217   identifier_node  strg: main      lngt: 4
@3218   statement_list   0    : @3222    1    : @3223
@3219   identifier_node  strg: __builtin_fork            lngt: 14
 ..........
@3222   bind_expr        type: @151      body: @3225
@3223   return_expr      type: @151      expr: @3226
@3224   function_decl    name: @3227     type: @3228     scpe: @176
                         srcp: <built-in>:0              chain: @3229
                         body: undefined                 link: extern
@3225   statement_list   0    : @3230    1    : @3231
@3226   modify_expr      type: @3        op 0: @3232     op 1: @2098
 ..........
                         link: extern
@3230   call_expr        type: @3        fn   : @3238    0    : @3239
@3231   return_expr      type: @151      expr: @3240
@3232   result_decl      type: @3        scpe: @3215     srcp: helloworld.c:3
                         note: artificial                size: @5
                         algn: 32

 ..........
@3238   addr_expr        type: @3243     op 0: @3159
@3239   nop_expr         type: @1932     op 0: @3244
@3240   modify_expr      type: @3        op 0: @3232     op 1: @2098
 ..........
@3243   pointer_type     size: @22       algn: 64        ptd : @3247
@3244   addr_expr        type: @3248     op 0: @3249
@3245   identifier_node  strg: __builtin_frob_return_addr
                         lngt: 26

 ..........
@3248   pointer_type     size: @22       algn: 64        ptd : @3255
@3249   string_cst       type: @3255     strg: Hello world!
  lngt: 14
```

The mystic TREE

Nodes

Types



Values

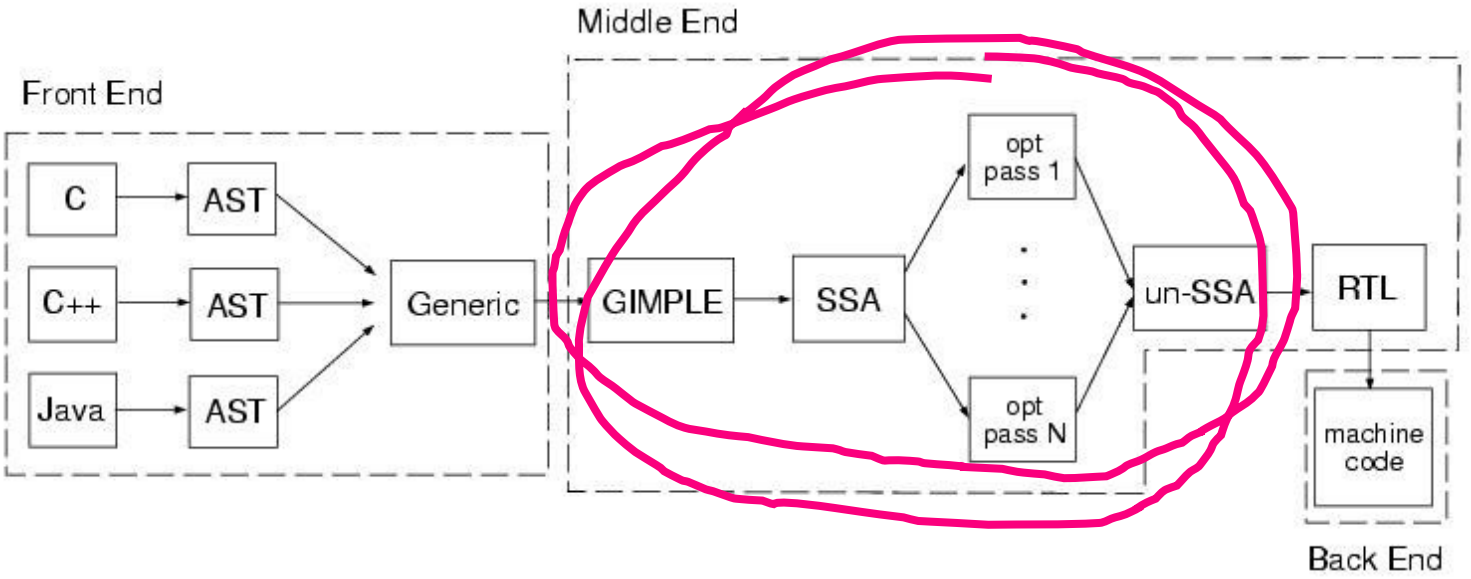
GENERIC

- Language-independent way of representing an entire function as trees
- Interface between parser and optimizers
- Superset of Gimple: imagine a language with a tree structure, similar to LISP
- Defined in gcc/tree.def

Important concepts:

Tree types and DECLs, expressions and statements

GIMPLE – A tree based representation



GIMPLE

The three address code

Target- and language independent optimization

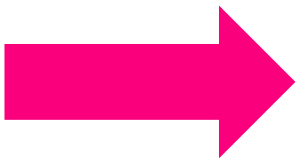
```
# Calculate one solution to the [[quadratic
equation]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x  := t9
```

https://en.wikipedia.org/wiki/Three-address_code

```c
int sub(void) {
    int x = 10;
    int y = 2;
    return x-y;
}

int add(void) {
    int a = 10;
    int b = 20;
    return a+b;
}

int main(void) {
    int d = add();
    int f = sub();

    int g = (d * 100 + 15) - (f * 10 - 50);
    return 0;
}
```

C



GIMPLE

```
sub ()
{
  int D.1809;
  int x;
  int y;

  x = 10;
  y = 2;
  D.1809 = x - y;
  return D.1809;
}


add ()
{
  int D.1811;
  int a;
  int b;

  a = 10;
  b = 20;
  D.1811 = a + b;
  return D.1811;
}
```

```
main ()
{
  int D.1813;

  {
    int d;
    int f;
    int g;

    d = add ();
    f = sub ();
    _1 = d * 100;
    _2 = _1 + 15;
    _3 = f * 10;
    _4 = _3 + -50;
    g = _2 - _4;
    D.1813 = 0;
    return D.1813;
  }
  D.1813 = 0;
  return D.1813;
}
```

GIMPLE in code

Instruction set and language structure much like any high level programming language

- GIMPLE_ASSIGN, GIMPLE_CALL, GIMPLE_RETURN, etc.
- GIMPLE_PHI, GIMPLE_ASM, etc.

Iterators & statement modifiers

Closely tied to TREE

Go-to tool for CFG and Tree SSA optimizers in GCC middle end

GIMPLE in code

```
// Iterating through basic blocks and Gimple sequences
FOR_EACH_BB_FN(bb, cfun) {

    for (gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi)) {

        gimple *statement = gsi_stmt(gsi);

        // Picking up on the printf within our helloworld.c
        if (gimple_code(statement) == GIMPLE_CALL){

            // Getting the first argument of printf
            tree arg = gimple_call_arg(statement, 0);

            // Building the new string argument
            tree satan = build_string(strlen("Hail Satan!!\n")+1, "Hail Satan!!\n");
            tree type = build_array_type(
                build_type_variant(char_type_node, 1, 0),
                build_index_type(size_int(strlen("Hail Satan!!\n"))));
            TREE_TYPE(satan) = type;
            TREE_CONSTANT(satan) = 1;
            TREE_READONLY(satan) = 1;
            TREE_STATIC(satan) = 1;

            // Replacing the helloworld string argument
            TREE_OPERAND(TREE_OPERAND((arg), 0), 0) = satan;
            gimple_call_set_arg(statement, 0, arg);
        }
    }
```

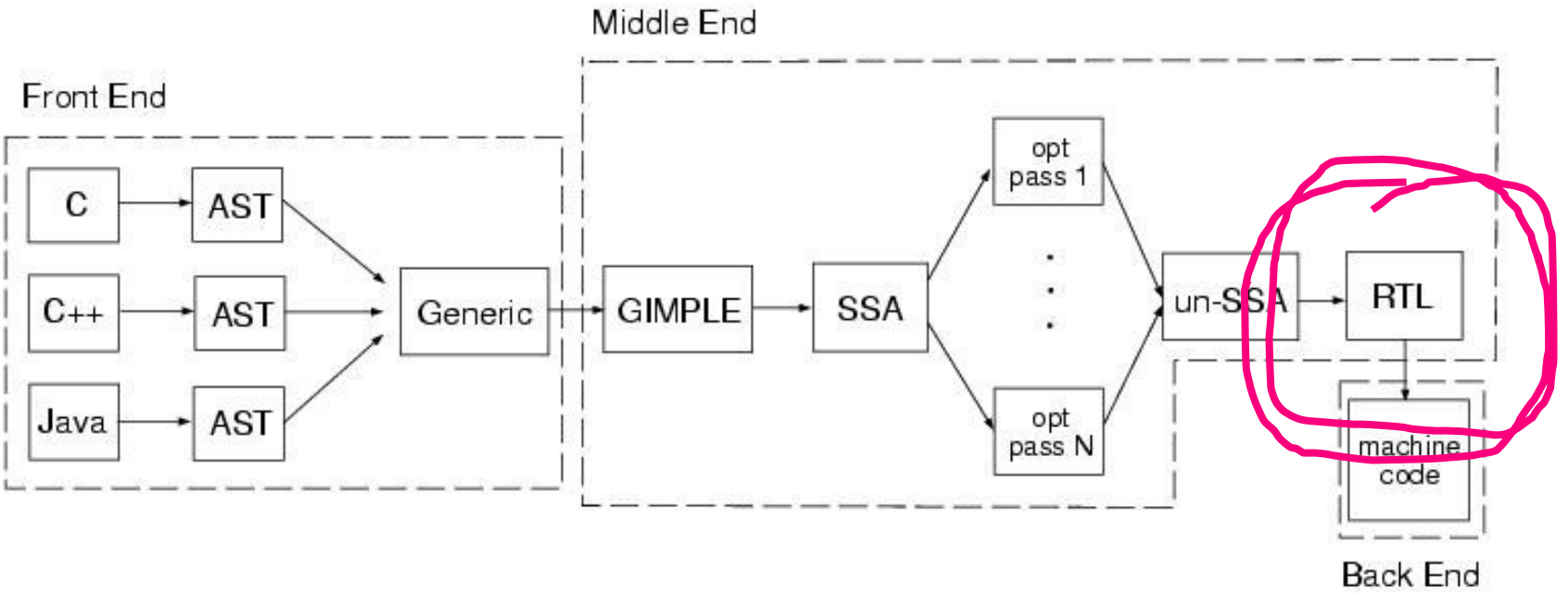
Iterator

Searching CALL statement

Building an argument

Modification

Register Transfer Language



RTL

RTL passes “implement” the machine definition

machine definition reflects the processor ABI

target dependent optimization

register allocation

machine code generation

rtl.def, rtl.h, <machine>.md

emit-rtl.h

“Assembly language for an abstract machine with infinite registers”

Instructions to be generated are described in an algebraic form that describes what the instruction does

The beauty lies within :)

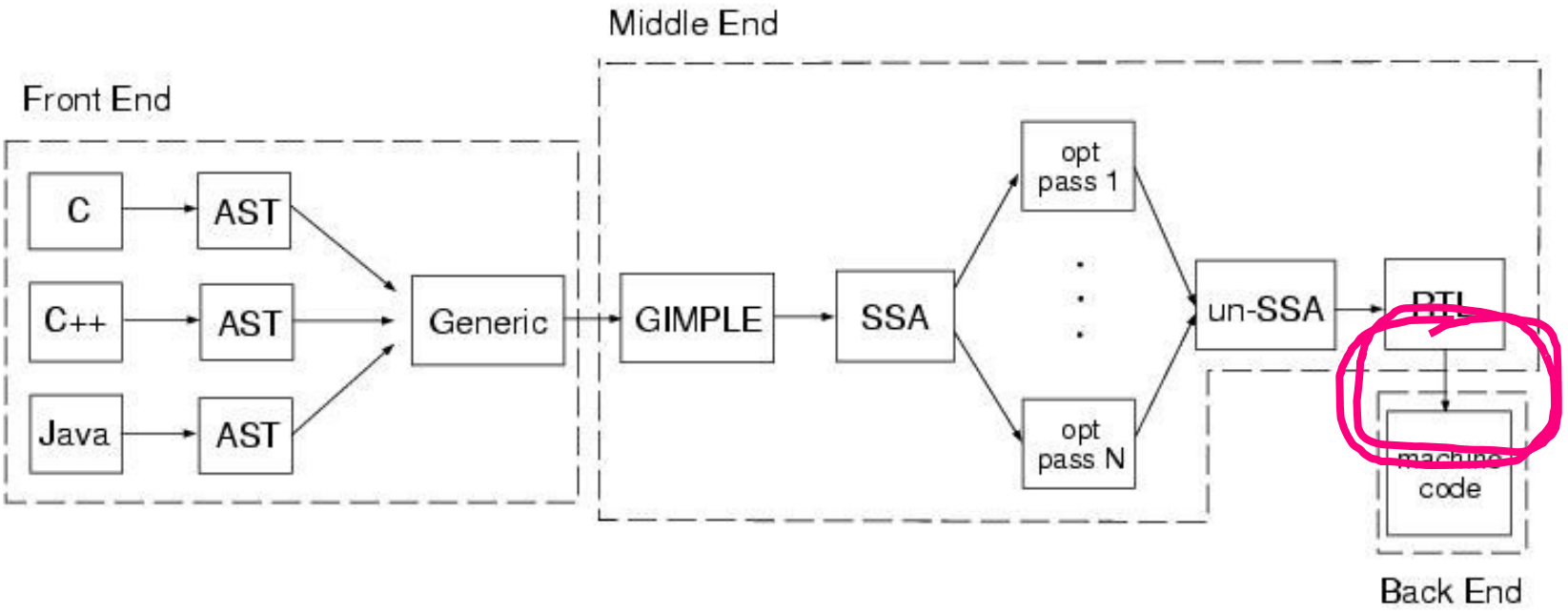
```
[...]
(insn 5 2 6 2
        (set (reg:DI 5 di)
        (symbol_ref/f:DI ("*.LC0") [flags 0x2]  <var_decl 0x7fd4f1a1ecf0 *.LC0>))
"helloworld.c":4 -1
(nil))

(call_insn 6 5 7 2 (set (reg:SI 0 ax)
        (call (mem:QI (symbol_ref:DI ("puts") [flags 0x41]
              <function_decl 0x7fd4f1974600 __builtin_puts>) [0 __builtin_puts S1 A8])
              (const_int 0 [0]))) "helloworld.c":4 -1
        (nil)
        (expr_list:DI (use (reg:DI 5 di))
(nil)))
[...]
```

```
// lea scratchReg, [memLocation + offset]
mySymbol= gen_rtx_SYMBOL_REF(Pmode, memLocation);
SYMBOL_REF_FLAGS(mySymbol) |= SYMBOL_FLAG_LOCAL;
leaInstruction = gen_rtx_SET(scratchReg, plus_constant(Pmode, mySymbol, offset));
emit_insn_before(leaInstruction, positionInsn);

// push variable (64 bit)
decrementStackP = gen_rtx_PRE_DEC(DImode, stack_pointer_rtx);
topOfStack = gen_rtx_MEM(DImode, decrementStackP);
pushInstruction = gen_rtx_SET(topOfStack, variable);
emit_insn_before(pushInstruction, positionInsn);

// mov scratchReg, sourceReg
movInstruction = gen_rtx_SET(scratchReg, sourceReg);
```

How do we generate RTL within GCC?

```
// call <location>


myInternalLabel = gen_label_rtx();
LABEL_NUSES(myInternalLabel)++;
ASM_GENERATE_INTERNAL_LABEL(LNAME, "L",
CODE_LABEL_NUMBER(myInternalLabel));


mySymbol = gen_rtx_SYMBOL_REF(Pmode, LNAME);
callInstruction = gen_rtx_CALL(Pmode, gen_rtx_MEM(FUNCTION_MODE,
mySymbol),    const0_rtx);
emit_call_insn_before(callInstruction, insn);



[...]
emit_label_before(myInternalLabel, insnAtLocation);
```

Machine Definitions <machine>.md



Machine Definitions <machine>.md

Main part of a gcc backend to be found in gcc/config/<machine>

i386.md

i386.opt

i386-modes.def

i386-protos.h

i386.c and i386.h

```
(define_insn "*sub<mode>_1"
   [(set (match_operand:SWI 0 "nonimmediate_operand" "=<r>m,<r>“)

      (minus:SWI
         (match_operand:SWI 1 "nonimmediate_operand" "0,0")
         (match_operand:SWI 2 "<general_operand>" "<r><i>,<r>m")))

      (clobber (reg:CC FLAGS_REG))]
   "ix86_binary_operator_ok (MINUS, <MODE>mode, operands)"
   "sub{<imodesuffix>}\t{%2, %0|%0, %2}"
   [(set_attr "type" "alu")
      (set_attr "mode" "<MODE>")])
```

IPA – Inter-Procedural Analysis

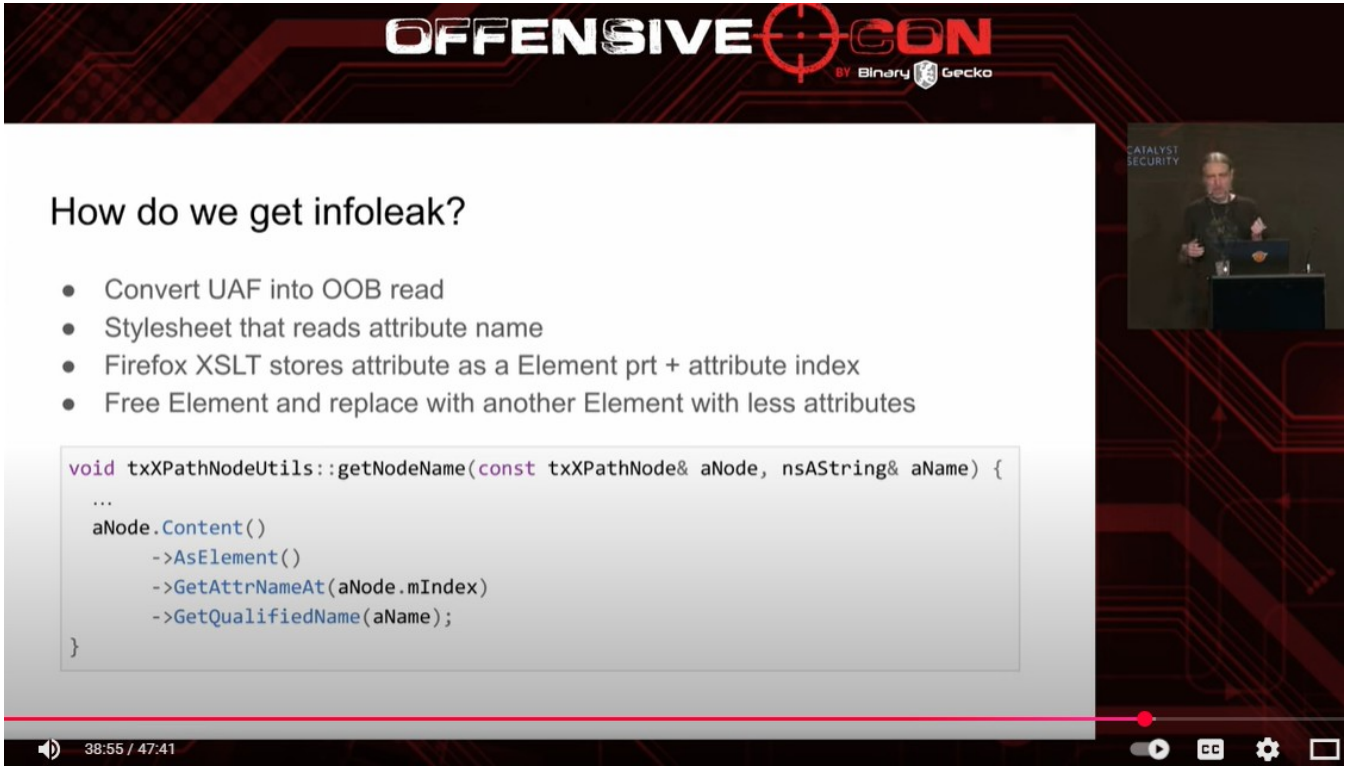
- IPA passes operate on the call graph and the varpool, inter-procedurally
- IPA_PASS and SIMPLE_IPA_PASS
- IPA LTO: stages partially run at compile time or at link time
- Go-to tools are essentially GIMPLE and GENERIC

generate_summary
write_summary
read_summary
execute
write_optimization_summary
read_optimization_summary
function_transform
variable_transform

But Security!!1!

Compiler Optimizations

… and what they have to do with security.


OFFENSIVE CON
BY Binary Gecko
How do we get infoleak?
Convert UAF into OOB read
Stylesheet that reads attribute name
Firefox XSLT stores attribute as a Element prt + attribute index
Free Element and replace with another Element with less attributes
void txXPathNodeUtils::getNodeName(const txXPathNode& aNode, nsAString& aName) {
...
aNode.Content()
->AsElement()
->GetAttrNameAt(aNode.mIndex)
->GetQualifiedName(aName);
}
CATALYST SECURITY
38:55 / 47:41
CC


OFFENSIVE CON
BY Binary Gecko
How do we ge
Convert UAF in
Stylesheet that
Firefox XSLT st
Free Element a
void txXPathNodeUti
...
aNode.Content()
->AsElement(
->GetAttrNam
->GetQualifi
}
38:55 / 47:41
OFFENSIVE CON
BY Binary Gecko
This shouldn't work…
const nsAttrName* AttrArray::GetSafeAttrNameAt(uint32_t aPos) const {
if (aPos >= AttrCount()) {
return nullptr;
}
return &mImpl->mBuffer[aPos].mName;
}
39:54 / 47:41


OFFENSIVE CON
BY Binary Gecko
How do we ge
Convert UAF in
Stylesheet that
Firefox XSLT st
Free Element a
void txXPathNodeUti
...
aNode.Content()
->AsElement(
->GetAttrNam
->GetQualifi
}
38:55 / 47:41
OFFENSIVE CON
BY Binary Gecko
This shouldn't wo
const nsAttrName* At
if (aPos >= AttrCo
return nullptr;
}
return &mImpl->mBu
}
39:54 / 47:41
OFFENSIVE CON
BY Binary Gecko
…and yet it does!
const nsAttrName* AttrArray::GetSafeAttrNameAt(uint32_t aPos) const {
if (aPos >= AttrCount()) {
return nullptr;
}
return &mImpl->mBuffer[aPos].mName;
}
The bounds check gets optimized away. Why?
Undefined behavior to the rescue!
41:40 / 47:41

Undefined Behavior

- There is a bounds check in the source, which got optimized away in the binary
- Ivan Fratric: “Why on earth would the compiler remove a bounds check?”
  - Bounds check returns nullptr
  - Calling function doesn’t check for nullptr, so if check fails that results in null pointer dereferentiation
  - Compiler knows that null pointer deref is undefined behavior; undefined behavior greenlights compiler to take remediative action, and compiler decided to remove the check entirely

Are there other such unicorns?

- OpenSSL "Memset Removal" Bug (CVE-2008-1196)
  - Cryptographic material wasn’t properly removed from memory after use since compiler thought operation was obsolete since the data wasn’t used after memset
- Clang/LLVM Stack Protector Misoptimization (CVE-2020-10771)
  - Clang removed stack protectors under certain conditions, eg. with tail calls
- Mozilla’s ARM64 Ion compiler in Firefox (CVE-2023-29548)
  - Wrong lowering (compiling from higher-level IR to lower level) of unsigned division when divisor is negation of power of two. The optimization incorrectly used absolute value logic
- Mbed TLS (versions prior to 3.6.4) (CVE-2025-52496)
  - Compiler removed security critical code due to a race condition in AESNI detection when certain compiler optimizations are applied, attackers can extract cryptographic keys or forge messages

Compiler Mitigations

- Stack Protectors
  - Canaries, Safe Stack, Shadow Stack, etc.
- Control Flow Integrity (CFI)
- CPU side channel mitigations
- Address Space Layout Randomization support (ASLR) through position independent code (PIC/PIE)
- And many others...

Spectre v2: Branch Target Injection

- Abuses indirect branch prediction, speculative execution and cache timing side-channels
- Tricks the CPU into speculatively executing memory it wouldn’t have executed otherwise, by poisoning indirect branch prediction
- If speculative execution leaves state behind in cache that state can be inferred using cache inference attacks
- This allows attackers to read privileged memory

https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

Spectre v2: Retpoline

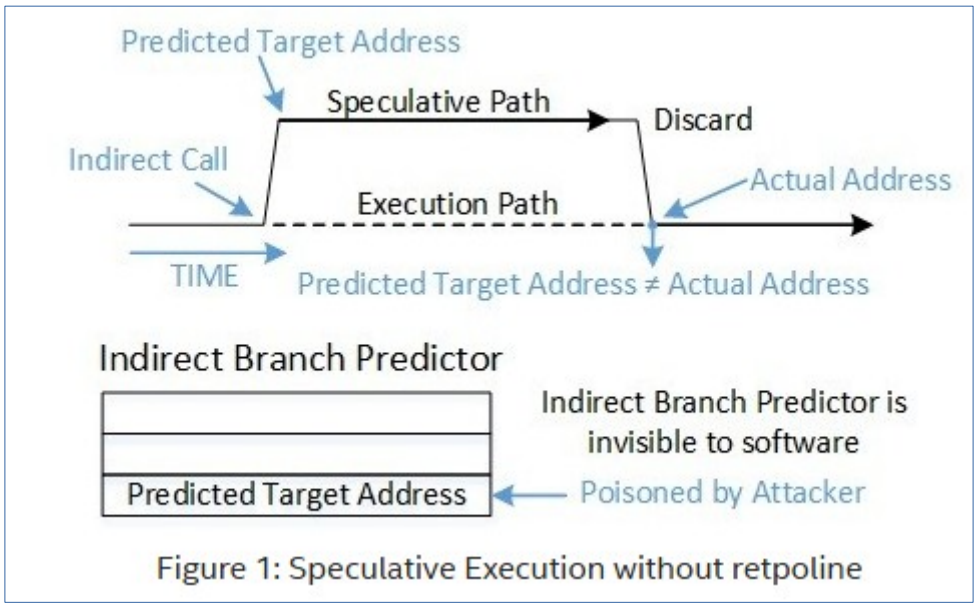


Figure 1: Speculative Execution without retpoline

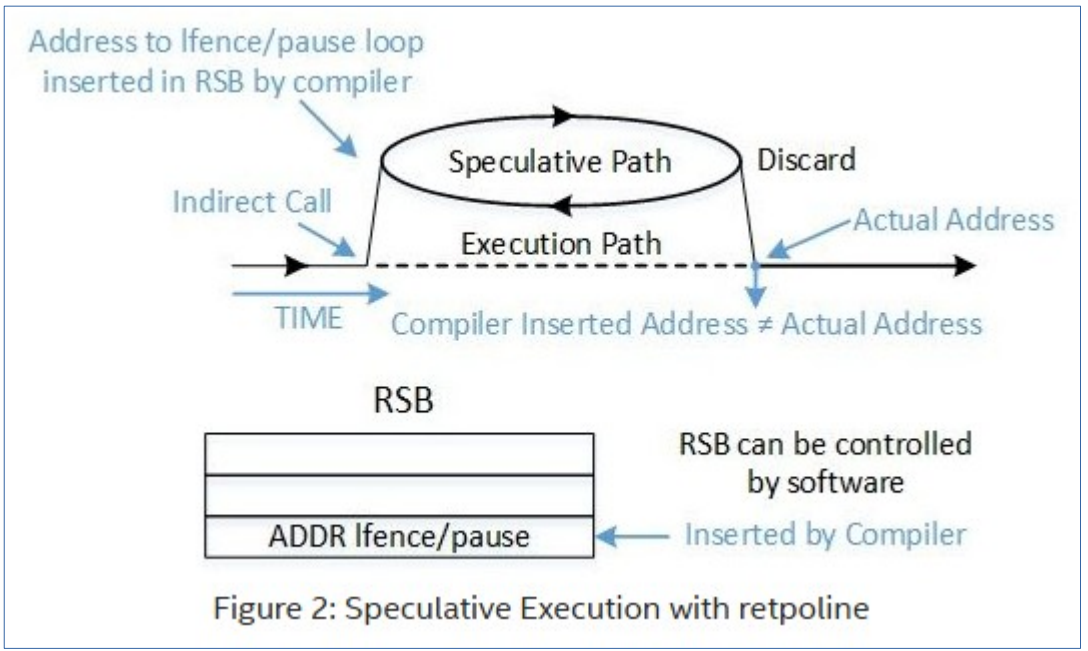


Figure 2: Speculative Execution with retpoline

https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html

Retpoline in GCC

Memory thunk where the function address is at the top of the stack:

```
__x86_indirect_thunk:
        call L2
L1:
        pause
        lfence
        jmp L1
L2:
        lea 8(%rsp), %rsp|lea 4(%esp), %esp
        ret

Indirect jmp via memory, "jmp mem", is converted to

        push memory
        jmp __x86_indirect_thunk

Indirect call via memory, "call mem", is converted to

        jmp L2
L1:
        push [mem]
        jmp __x86_indirect_thunk
L2:
        call L1
```

Register thunk where the function address is in a register, reg:

```
__x86_indirect_thunk_reg:
        call    L2
L1:
        pause
        lfence
        jmp     L1
L2:
        movq    %reg, (%rsp)|movl    %reg, (%esp)
        ret

where reg is one of (r|e)ax, (r|e)dx, (r|e)cx, (r|e)bx, (r|e)si, (r|e)di,
(r|e)bp, r8, r9, r10, r11, r12, r13, r14 and r15.

Indirect jmp via register, "jmp reg", is converted to

        jmp __x86_indirect_thunk_reg

Indirect call via register, "call reg", is converted to

        call __x86_indirect_thunk_reg
```

https://github.com/gcc-mirror/gcc/commit/da99fd4a3ca06b43b08ba8d96dab84e83ac90aa7#diff-806048569684df6eeab0a325b82fdd48edf5314368e8e4518c7522a5c358fe63

Software Protections

- Source level (Developer)
- IR-level (Compiler)
- Machine code / post link stage (aka. Runtime packers)

Compiler Based Protections

- Control flow flattening
- Basic block splitting / reordering
- Bogus control flow / instructions
  - Opaque branches, bloat code, etc.
- String encryption, constant encryption
- Replace simple arithmetic with complex operations
- Breakpoint / hook / debugger detections
- Function inlining/outlining
- Instruction substitution
- SO MUCH MORE

Why Obfuscation Matters

```
.text:00401047
.text:00401047                          loc_401047:                             ; CODE XREF: .text:loc_401047↑j
.text:00401047 0F 84 FF FF FF FF        jz      near ptr loc_401047+5
.text:0040104D 00 68 43                 add     [eax+43h], ch
.text:00401050 22 15 90 58 31 05        and     dl, ds:5315890h
.text:00401056 00 30                    add     [eax+0], dh
.text:00401058 40                       inc     eax
.text:00401059 00 B9 00 00 00 10        add     [ecx+10000000h], bh

.text:00401045 3B C0                    cmp     eax, eax
.text:00401045                          ; ---------------------------------------------------------------------------
.text:00401047 0F                       db 0Fh
.text:00401048 84 FF FF FF              dd 0FFFFFF84h
.text:0040104C                          ; ---------------------------------------------------------------------------
.text:0040104C FF 00                    inc     dword ptr [eax]
.text:0040104E 68 43 22 15 00           push    00152243h
.text:00401053 58                       pop     eax
.text:00401054 31 05 00 30 40 00        xor     dword_403000, eax
.text:0040105A B9 00 00 00 10           mov     ecx, 10000000h
```

Obfuscation Passes

For Developers

The items in this section describe the different obfuscation passes available in O-MVLL. The documentation of the passes provides information about when and how to use the obfuscation as well as its limitations.

There is also an overview of the pass synthesized by the following items:

Protection

- **Static**: the pass aims at protecting against static code analysis (decompilation, disassemblers, ...)
- **Dynamic**: the pass aims at protecting against dynamic code analysis (hooking, instrumentation, debugging, ...)

Resilience

This property tries to give a level of strength against a fully automated attack. If such an attack exists, it is mentioned in the *References* section of the pass.

Obfuscation Overhead

The kind of overhead introduced in the program when using the given obfuscation pass. The values can be:

- **Code size**: the assembly code is larger.
- **Data size**: the raw data of the binary is larger.
- **Memory size**: the size of the binary in memory is larger [1].

Thank you

Marion Marschalek
pinkflawd@gmail.com

Resources

https://code.woboq.org/gcc/gcc/

https://gcc.gnu.org/onlinedocs/gccint/index.html

https://github.com/enferex/sataniccanary/

https://github.com/ephox-gcc-plugins

https://medium.com/@prathamesh1615/adding-peephole-optimization-to-gcc-89c329dd27b3

https://www.airs.com/dnovillo/200711-GCC-Internals/200711-GCC-Internals-7-passes.pdf

https://www.mitre.org/sites/default/files/publications/supply-chain-attack-framework-14-0228.pdf

https://lwn.net/Articles/457543/

https://www.cse.iitb.ac.in/grc/slides/cgotut-gcc/topic8-retarg-mode.pdf

https://www.cse.iitb.ac.in/~uday/courses/cs715-09/gcc-rtl.pdf

https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

https://codesynthesis.com/~boris/blog/2010/05/03/parsing-cxx-with-gcc-plugin-part-1/

https://kristerw.blogspot.com/2017/08/writing-gcc-backend_4.html

https://www.usenix.org/sites/default/files/conference/protected-files/kemerlis_usenixsecurity12_slides.pdf

ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf

https://pdfs.semanticscholar.org/cafc/c15a1602c5a8090606333b3bdb42e9e80654.pdf

https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html

https://www.youtube.com/watch?v=U1kc7fcF5Ao