From Debugging-Information Based Binary-Level Type Inference to CFG Generation

Dongrui Zeng
Pennsylvania State University
State Collge, PA, USA
dongrui.zeng@gmail.com

Gang Tan
Pennsylvania State University
State Collge, PA, USA
gtan@cse.psu.edu

ABSTRACT

Binary-level Control-Flow Graph (CFG) construction is essential for applications such as control-flow integrity. There are two main approaches: the binary-analysis approach and the compiler-modification approach. The binary-analysis approach does not require source code, but it constructs low-precision CFGs. The compiler-modification approach requires source code and modifies compilers for CFG generation. We describe the design and implementation of an alternative system for high-precision CFG construction, which still assumes source code but does not modify compilers. Our approach makes use of standard compiler-generated meta-information, including symbol tables, relocation information, and debugging information. A key component in the system is a type-inference engine that infers types of low-level storage locations such as registers from types in debugging information. Inferred types enable a type-signature matching method for high-precision CFG construction.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

Control flow graphs; type inference; debugging information

ACM Reference Format:
Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3176258.3176309

1 INTRODUCTION

A program's Control-Flow Graph (CFG) is a representation of the program's possible control flows. Constructing CFGs for binary programs is a necessary step for many security applications. A prominent example is Control-Flow Integrity (CFI [1]), which enforces a binary-level CFG on a program to mitigate control-flow hijacking attacks such as Return-Oriented Programming [25]. Another example is binary-level program analysis (e.g., [4, 9, 24, 26, 34, 35]) including symbolic execution, taint analysis, program slicing, testing-suite generation, and liveness analysis, which all require the CFG of a binary program to operate. Finally, binary rewriting (e.g., [1, 29, 31]) requires CFGs to rewrite code for security.

The main difficulty of constructing binary-level CFGs is to compute control-flow targets for indirect branches, which include return instructions, indirect jumps (jumps via register or memory operands), and indirect calls (function calls via register or memory operands). Since indirect-branch targets depend on values that are computed at runtime, it is challenging to predict them statically.

Researchers have explored two main approaches for binary-level CFG construction: the binary-analysis approach and the compiler-modification approach. The *binary-analysis* approach [19, 28, 36, 37] analyzes binary code and its contained information for CFG construction and has the benefit of not requiring source-code availability. However, the drawback is that its constructed CFGs are of low precision, for the reason that binary code lacks structured information that helps disassembly and compute accurately the target sets of indirect branches [2]. Consequently, its constructed CFGs contain many spurious edges and may miss edges. The precision of CFGs is critical for some security applications. In CFI, a lower-precision CFG allows the attacker greater freedom over how control flows can be manipulated. It has been shown that coarse-grained CFI (which enforces low-precision CFGs) is much easier to attack than fine-grained CFI [8, 10, 12].

Another way is to use compilers for CFG construction. Existing compilers construct CFGs of programs for the purpose of program analysis and optimization. However, CFGs constructed by compilers are at the level of Intermediate Representations (IR). After IR code is translated to assembly code (which is further optimized through architecture-specific optimizations), the IR-level CFG cannot be directly transferred to the low level. Furthermore, the IR-level CFGs by most compilers are often incomplete, especially for those inter-procedural edges resulting from indirect calls and returns.

To address these problems, the *compiler-modification* approach for binary-level CFG generation[20, 23, 27, 30] modifies existing compilers to propagate information from source to binary code and then performs CFG construction directly at the binary level. For instance, while standard compilers throw away type information during compilation, MCFI [20] modifies LLVM 3.5 to preserve type information in binaries and implements a type-signature approach for binary-level CFG generation. Forward-Edge CFI [27] modifies GCC 4.9 to enforce a form of CFI that protects virtual calls in C++ applications. This approach constructs CFGs of much higher precision than the binary-analysis approach. However, it requires source code. Furthermore, the practical limitation of modifying a compiler is that the CFG-construction code has to be upgraded when the compiler is upgraded. When a compiler migrates to a newer version, its

CODASPY '18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5632-9/18/03…$15.00
https://doi.org/10.1145/3176258.3176309

internal API often changes, meaning any passes that were added to an older version have to be changed to use the new API; this is not an easy task. Upgrading MCFI to the latest LLVM (version 5.0) and Forward-Edge CFI to the latest GCC (version 7.2) would be a major engineering undertaking. In the case when a CFI implementation (e.g., Forward-Edge CFI in LLVM) has been upstreamed to the main branch of the compiler, it does force compiler developers to migrate the CFI implementation with compiler upgrades; however, it can be laborious for compiler developers. Moreover, CFG construction in one compiler cannot easily be reused for a different compiler.

In this paper, we describe the design and implementation of an alternative approach for high-precision CFG construction, without compiler modification. The key observation is that compilers already generate a collection of standard meta-information such as debugging information that can be reused for CFG construction. Therefore, our system takes binary code with standard meta-information as input, performs the key step of type inference at the binary level, and uses the recovered types for CFG construction.

Meta-information used in our system includes symbol tables, relocation information, and debugging information; they are designed for purposes such as debugging and standard compilers generate and maintain them across upgrades. Our approach still requires source code for meta-information generation. But it has the benefit of not modifying compilers, meaning that the implementation can largely be reused across different compiler versions and even across compilers. Moreover, our experiments show that the precision of CFGs generated by our implementation is comparable to the precision of CFGs generated by those that do modify compilers. Major contributions of our system are the followings:

- As far as we are aware, our system is the first one that constructs CFGs from standard meta-information generated by compilers. Compared to the approach of modifying compilers, most of our system can be reused across compiler versions and across compilers. Compared to the binary-analysis approach, our system generates CFGs of much higher precision, comparable to those systems that modify compilers.
- The key step in our CFG construction is a general binary-level type-inference procedure. Flow-based constraints are generated from binary code together with meta information and are solved to infer types of registers and stack locations. From type-inference results, a high-precision CFG is constructed following the type-signature approach for CFG construction.
- We have performed thorough experiments to evaluate the effectiveness of type inference and the precision of generated CFGs, and validated our system with different optimization levels used by compilers, multiple compiler versions, and multiple compilers (GCC and LLVM).

2 BACKGROUND

Standard compiler-generated meta-information includes symbol tables, relocation information, and debugging information. We next briefly describe the information contained in each kind.

Symbol tables. Compilers generate information about symbols (e.g., function and variable names) from source code in the form of symbol tables and embed the tables in binaries. Tools such as linkers and debuggers consume symbol tables to locate and relocate a program's symbolic definitions for static/dynamic linking and debugging. Symbol tables contain entries for symbols and each entry stores a name, a binding address, the type of the symbol, and other information. Our system compiles source code to generate unstripped binaries, which have the full symbol tables.

Relocation information. Before linking, memory addresses of compilation units such as functions and global data are unknown. Compilers therefore generate relocation entries so that static/dynamic linkers can patch the program's code and data after memory addresses become known. The patch process is for relocating code and data in object code and is crucial for separate compilation.

Debugging information. There are several formats for debugging information including STABS [16] and DWARF [11]. We focus on DWARF since it is the standard format adopted by most compilers including GCC and LLVM and also most debuggers.

The DWARF format is well-designed for debugging and includes several kinds of debugging information: (1) information in source code such as types and scope of identifiers is included in the form of *Debugging Information Entries (DIEs)*; (2) line-number information is included for recording the correspondence between binary and source code; (3) *location descriptions* are included for describing storage locations of variables during execution; (4) the *Canonical Frame Address (CFA)* information is included for describing the stack layout; this information is used during debugging and can also be useful for producing back traces when exceptions are raised. More details about debugging information, especially types, will be discussed in Sec. 4.1.

Incomplete and inaccurate meta-information. Symbol tables and relocation information are critical to static and dynamic linkers and therefore it is reasonable to assume that they are complete and accurate. The picture is different for debugging information, however. First, while all compilers generate a basic set of debugging information, the generation of advanced debugging information is compiler dependent. For example, GCC generates call-site information, which tells whether calls are tail calls and the types of arguments, while LLVM does not produce such information.

Second, debugging information may be inaccurate, especially in optimized code. It is well known that line-number information becomes unusable in optimized code: after optimizations such as code motion, it is often not possible for the compiler to update the relationship between binary and source code in terms of line numbers. Another example is that the CFA information for describing stack layouts may become inaccurate after compiler optimizations.

Inaccurate debugging information is a concern for our system as this can lead to wrong CFGs being constructed. Our decision is to not use such information, including line-number information and stack-layout information. Our CFG construction relies on type information, scope information, and location descriptions in debugging information; these kinds of information are found to be accurate across compilers, even after optimizations. Incomplete debugging information is less of a concern in that our system uses only what is available in debugging information to refine CFGs so that incomplete information just results in less-precise CFGs.

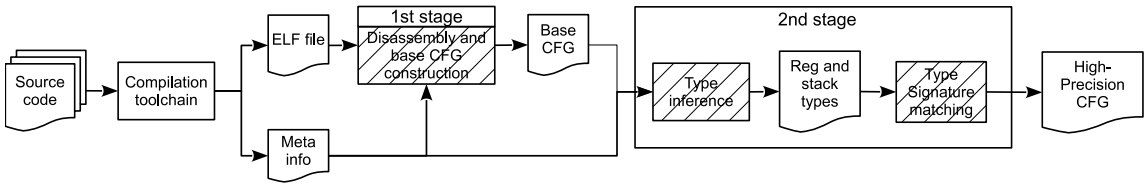


Figure 1: System architecture for high-precision CFG construction.

3 OVERVIEW

Fig. 1 presents the architecture of our CFG-construction system. The system takes source code, which is fed to an unmodified compilation toolchain (a compiler and a linker) to generate a binary program with standard meta-information. It then constructs the binary program's CFG in two stages.

In the first stage, a recursive-traversal disassembly algorithm is applied to disassemble the binary program and construct a coarse-grained CFG, which is called the base CFG in the figure. The first stage also takes as input compiler-generated meta-information, which tells the entries of all functions and makes disassembly complete. Its generated CFG is coarse grained in that many targets are allowed for indirect branches. On the other hand, control-flow edges out of direct branches (gotos, if conditionals, and direct calls) are accurate because their targets can be statically computed.

The second stage takes the base CFG and the meta information as input and refines the control-flow targets of indirect branches. Its key component is a type-inference engine that infers types of storage locations from types in debugging information. From the inferred types, the second stage follows a type-signature approach [20, 27] to narrow down the target list of an indirect branch.

Following the two-stage design, we have built a prototype system that constructs high-precision CFGs for binaries that are generated from different compilers (GCC and LLVM) and different compiler versions. The prototype is for Linux x86-32 binaries in the ELF format.[1] Further, the prototype assumes that the source code is in the C language because its type-inference engine infers C-like types for storage locations. Previous work [13, 21] has shown that the type-signature approach applies equally well on C++'s type system with the help of Class Hierarchy Analysis (CHA). This work and recent work on C++ CFG construction [22] give us confidence that our meta-information based approach can be generalized to programs in C++, with more work on type inference.

As a high-level note, our approach is compiler independent since the meta information relied on by our system (symbol tables, relocation and debugging information) all have standard formats that are independent of compilers; in particular, ELF and DWARF. As long as a compiler supports those standard formats, the parsing and use of meta information are compiler independent.

4 TYPE INFERENCE

The key component in our CFG-building framework is a generic type-inference system that infers types of storage locations (registers and stack slots) from source-level types included in debugging information. The type-inference engine works on one function at a time, starting from types in debugging information (which tell at least the entry types of parameters and local variables) and inferring the types of storage locations in the middle of the function.

It has four major components: (1) a component that collects types from debugging information; (2) a stack-layout inference algorithm that normalizes the representation of stack slots with respect to a canonical frame address; (3) a constraint-generation component that turns instructions and control flows into type constraints; (4) a constraint-solving component that solves constraints and computes a set of types for storage locations at every code address in the function. We next discuss these components in detail.

4.1 Debugging type information

Type inference starts from those types already included in debugging information. We next describe in detail what type information is there in debugging information. At a high level, types of source-code identifiers are included. What types of information are attached depends on the kinds of identifiers.

Functions. For a function, debugging information contains a debugging entry for the function, followed by entries for the function's formal parameters and local variables. From the entry of the function and the entries of its formal parameters, we can know the function's number of parameters, the parameter types, the return type, and also the range of code addresses of the function's body.

Formal parameters and local variables. These are function-local identifiers. The debugging entry for such an identifier contains the type of the identifier as well as a location description that describes where the identifier is stored.

A location description $[(st_1, rng_1), (st_2, rng_2), \ldots, (st_n, rng_n)]$ is a list of pairs, where st_i is a storage location, which is either a register (such as `eax` in x86-32), a stack location, or a location in global data sections (which store global variables), and rng_i is a code-address range. The above location-description list is interpreted as follows: the identifier is stored in location st_i before any instruction whose address is within address-range rng_i.

Fig. 2 presents an example. In this and other examples of the paper, for clarity we will use a pseudo-assembly syntax whose notation is described as follows. We use "`:=`" for an assignment (that is, a move instruction). When an operand represents a memory operand, the memory address is put into `mem(-)`. For instance, `mem(esp)` is a memory operand with the address in `esp`, while `esp` itself represents a register operand. We will also use the syntax "`if ... goto ...`" for conditional jumps (on x86, it represents a comparison followed by a jump instruction).

The example in Fig. 2 includes the skeleton code for a function. Assume there is a local variable named `i` in the function and it is initially allocated on the stack; specifically, `i` is stored at the top of

[1] The prototype relies on a previous formal model of x86-32 for decoding [17]. That model lacks the decoding support for x86-64. We are in the process of extending it to cover the 64-bit.

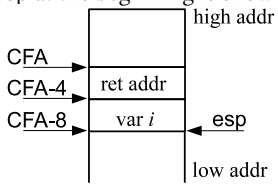
```
// Assume variable i is initially at the top of the stack
addr1:  ...
        ...
addr2:  eax := mem(esp)
        ...
addr3:  ebx := eax
        ...
addr4:  ret
```

Debugging entry for `i` :
`int; [(CFA-8,[addr1,addr2]), (eax,[addr2+1,addr3]), (ebx,[addr3+1,addr4])]`

Figure 2: An example for debugging entries.

the stack. At addr2, the storage location of i is moved to register eax and at addr3 the location is moved to ebx.[2] The debugging entry for *i* is also shown in the figure (for brevity, we have omitted information irrelevant for our discussion, such as line numbers). It tells the source-code type of the variable and its location description, which means that the variable is stored in stack slot "CFA-8" between addr1 and addr2, in eax between addr2+1 and addr3, in ebx between addr3+1 and addr4.

The representation of a stack slot is normalized relative to CFA (Canonical Frame Address). The CFA address is typically defined to be the value of the stack pointer at the call site that invokes the current function. Therefore, the return address is stored at "CFA-4" since the return address is pushed by the call instruction (and the stack grows from high to low memory addresses). When local variables are allocated on the stack, their addresses are expressed relative to CFA in location descriptions. For the example, the relation between CFA and esp at the beginning is shown as follows:



The types and location descriptions in debugging entries for local variables (and parameters) tell the types of some storage locations at specific code addresses. For the example in Fig. 2, the debugging entry for i tells that int is the type of stack slot cfa-8 between addr1 and addr2, the type of eax between addr2+1 and addr3, and the type of ebx between addr3+1 and addr4.

Global variables. Similar to local-variable entries, debugging entries for global variables tell their types and location descriptions. However, a global variable's storage location does not change over the course of program execution and therefore its location description has only one pair [(*st*, *rng*)], where *st* encodes the memory address where the global variable is stored in global data sections, and *rng* is either the entire range of code addresses or a portion of it depending on whether the global variable is external or static.

[2]In unoptimized code, a local variable is stored in a specific stack slot for the entire function; however, code produced by higher-optimization levels can move the storage of a variable to a register to improve the efficiency of accessing the variable.

Incomplete type information. As we have shown, debugging information tells the types of some storage locations at certain code addresses. However, types in debugging information are for source-level constructs and, when source code is compiled to binary code, compilers are conservative in embedding types in debugging information, resulting in incomplete type information. As an example in Fig. 2, the location description does not tell the type of eax between addr3+1 and addr4, even though the type should remain the same before eax is modified. As another example, suppose a local variable is stored in eax initially and is then copied to ebx, and the location description states that the variable is stored in eax even after copying; then ebx's type after copying should be the same as eax, even though debugging information does not tell it directly. Sec. 6 will show that for large programs types can be missing in debugging information for over 50% of indirect-call operands.

4.2 Stack layout inference

As presented earlier, stack slots in debugging information are normalized and encoded as offsets to CFA. A binary program accesses stack slots, however, via memory addresses in the form of offsets to registers such as esp and ebp. To be able to track and infer types of stack slots, we must infer the relationship between registers and CFA. As a simple example, suppose the program reads the stack at address "esp" and debugging information tells us that the value at stack slot "CFA-8" is of type *t*; we cannot know the type of the value at address "esp" unless it is given that esp=CFA-8.

Some compilers generate in debugging information call-frame information that encodes the relation between CFA and register values. However, it is often incomplete and sometimes inaccurate. For instance, LLVM generates call-frame information for function prologues but such information is not generated for code after prologues. GCC's call-frame information may become inaccurate after code is optimized, for example, when code is moved after return instructions during optimization. Therefore, we have built a static analysis that infers the relation between registers and CFA; we call it stack layout inference.

The static analysis takes the assembly code of a function and its base CFG as input and at every address builds a set of equations of the following form: $\{r_1 = \texttt{CFA} + off_1, ..., r_n = \texttt{CFA} + off_n\}$. If a register does not point to the stack, then no equation for the register is included in the above set.

At the beginning of the function, the analysis assumes "{esp=CFA-4}" (the four bytes between CFA and esp are the return address). The transfer function for an instruction is built straightforwardly based on the instruction's semantics. Fig. 3 provides some examples: after "push ebp", we have "{esp=CFA-8}"; after another instruction "ebp:=esp", we get "{esp=CFA-8, ebp=CFA-8}".

At a control-flow merge point, the merge function is defined to be the intersection of the sets of equations from the paths being merged. In Fig. 3, address 7 is both a destination of the if-test at address 5 and the fall-through destination of the instruction at address 6. Therefore, we take the intersection of "{esp=CFA-20, ebp=CFA-8}" and "{esp=CFA-20, ebp=CFA-8, ebx=CFA-20}".

For a function call, the stack layout remains the same after the call; it assumes a function call preserves the stack layout. If the

```
     {esp=CFA-4}
1    push ebp
     {esp=CFA-8}
2    ebp := esp
     {esp=CFA-8, ebp=CFA-8}
3    push ebx
     {esp=CFA-12, ebp=CFA-8}
4    esp := esp - 8
     {esp=CFA-20, ebp=CFA-8}
5    if  eax==0 goto 7
     {esp=CFA-20, ebp=CFA-8}
6    ebx := esp
     {esp=CFA-20, ebp=CFA-8, ebx=CFA-20}

     {esp=CFA-20, ebp=CFA-8}
7    esp := esp + 8
     {esp=CFA-12, ebp=CFA-8}
8    pop ebx
     {esp=CFA-8, ebp=CFA-8}
9    pop ebp
     {esp=CFA-4}
10   ret
```

Figure 3: Stack layout inference for a toy function.

$$
\begin{array}{rcl}
st & := & r \mid \texttt{CFA} + off \\
x, y, z & := & st_l^- \mid st_l^+ \mid gl \\
C & := & x \supseteq y \mid x \supseteq *(y + off) \mid x \supseteq \&(y + off) \\
\\
t & := & \top \mid \text{void} \mid \text{int} \mid \text{char} \mid \text{float} \mid \text{double} \mid t* \mid t[n] \mid \\
 & & (t_1, \ldots, t_n) \rightarrow t \mid \{id_1 : t_1, \ldots, id_n : t_n\} \mid \text{tbl\_ent} \\
X, Y, Z & := & x \mid x : t \\
D & := & X \supseteq Y \mid X \supseteq *(Y + off) \mid X \supseteq \&(Y + off)
\end{array}
$$

Figure 4: Syntax of type constraints.

function has a loop, a standard worklist algorithm is used to calculate a fixed point; the worklist terminates since the underlying lattice is of finite height.

With the result of stack layout inference, accesses to the stack via registers plus constants can be normalized with respect to CFA.

4.3 Constraint generation

At a high level, constraints specify relations between types of storage locations based on how values flow in the input function. Our system tracks the types of registers, stack slots, and locations in global data sections. It does not explicitly track the types of heap locations that are dynamically allocated; however, values read from memory can still be assigned types in some cases. For instance, if eax holds a pointer to a dynamically allocated struct and debugging information tells that eax's type is a struct-pointer type, then a memory read via eax plus a constant offset returns a value whose type can be inferred from the struct type and the offset.

Fig. 4 presents the syntax of type constraints. We use *st* for a storage location, which is either a register or a stack slot in the form of CFA plus some offset. We use *gl* for the location of a global variable (which resides in global data sections). We separate *gl* from *st* since the type of a global variable does not change over the course of program execution (further, debugging information tells the type); in contrast, types in registers and stack slots may change over program execution.

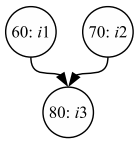
Code	Constraints
10: eax:=ebx	$\texttt{eax}_{10}^+ \supseteq \texttt{ebx}_{10}^-$ $\forall st \neq \texttt{eax},\ st_{10}^+ \supseteq st_{10}^-$
// assume ebp=CFA-8 20: eax:=mem(ebp+12)	$\texttt{eax}_{20}^+ \supseteq (\texttt{CFA} + 4)_{20}^-$ $\forall st \neq \texttt{eax},\ st_{20}^+ \supseteq st_{20}^-$
// ebx unrelated to CFA 30: eax:=mem(ebx+4)	$\texttt{eax}_{30}^+ \supseteq *(\texttt{ebx}_{30}^- + 4)$ $\forall st \neq \texttt{eax},\ st_{30}^+ \supseteq st_{30}^-$
// assume ebp=CFA-8 40: eax:=lea(ebp+12)	$\texttt{eax}_{40}^+ \supseteq \&(\texttt{CFA} + 4)_{40}^-$ $\forall st \neq \texttt{eax},\ st_{40}^+ \supseteq st_{40}^-$
// ebx unrelated to CFA 50: eax:=lea(ebx+4)	$\texttt{eax}_{50}^+ \supseteq \&(\texttt{ebx}_{50}^- + 4)$ $\forall st \neq \texttt{eax},\ st_{50}^+ \supseteq st_{50}^-$
	$\forall st,\ st_{80}^- \supseteq st_{60}^+$ $\forall st,\ st_{80}^- \supseteq st_{70}^+$

Figure 5: Examples for constraint generation.

We use x, y, and z for type variables. They can stand for the set of types of registers or stack slots at a particular point, or the type of a global variable. In particular, we use st_l^- to stand for the type set of storage location st before the instruction at address l, and st_l^+ for the type set of storage location st after the instruction at address l. For instance, $\texttt{eax}_{10}^-$ stands for the type set of eax before instruction at address 10 and $\texttt{eax}_{10}^+$ for the type set of eax afterwards. Since the type of a global variable does not change, *gl* is used to stand for the type of the corresponding global variable.

Type constraints are of three forms. Value-flow constraint "$x \supseteq y$" models the case when y's value flows to x; as a result, x's type set should be a superset of y's type set. A dereference-flow constraint "$x \supseteq *(y + off)$" models the case when the value at address $y + off$ is read from memory and flows to x; address $y + off$ is assumed to not point to a stack slot, since stack slots are represented using CFA plus an offset. An address-flow constraint "$x \supseteq \&(y + off)$" models the case when the memory address of $y + off$ flows to x.

Constraints are generated conservatively from instructions. Due to space limitation, we cannot enumerate all the cases. Instead, we just discuss the cases for typical instructions and illustrate them via examples in Fig. 5. For a register-to-register move instruction $r_1 := r_2$, constraints are generated to express that the type set of r_1 afterwards is a superset of r_2 beforehand (since r_2's value flows to r_1) and the type set of other storage locations afterwards is a superset of the same storage locations beforehand since their values are unchanged (for brevity, we will omit the mentioning of constraints concerning unchanged storage locations in the rest of the discussion). An example of "eax:=ebx" is in Fig. 5.

For a memory-to-register move $r_1 := mem(op)$, constraint generation depends on operand op. Suppose $op = r_2 + off$. If r_2 points to

the stack before the move instruction (determined by stack layout analysis), then a constraint is generated to state that r_1's type set afterwards is a superset of the type set of the corresponding stack slot beforehand. An example of "`eax:=mem(ebp+12)`" is in Fig. 5. If r_2 does not point to the stack, then a dereference-flow constraint is generated to relate r_1 and r_2. An example of "`eax:=mem(ebx+4)`" is included in Fig. 5. When *op* is an immediate value, our system checks whether it is a memory address that holds a global variable. If it is, a value-flow constraint states that r_1's type set is a superset of the global variable's type. If it is not, no constraint is generated for r_1 after the move; this is an approximation, an unconstrained type variable will be assigned the $\top$ type, meaning no information is there for its type. In x86, an operand *op* can also use other complex addressing modes (such as a base register plus a register times a scalar and a displacement); in these cases, we approximate by not generating constraints for r_1.

x86 also has `lea` (load effective address) instructions, which move memory addresses but do not perform memory reads. Fig. 5 shows two examples. In example one, "`eax:=lea(ebp+12)`" moves the address of `ebp+12` into `eax`. Assuming `ebp=CFA-8`, the instruction moves the address of stack storage location `CFA+4` to `eax`; we use an address-flow constraint to model that. The second example is similar, except it moves a heap address; it is also modeled by an address-flow constraint.

Constraints generated for a register-to-memory move $mem(op) := r_1$ also depend on the shape of *op*. If $op = r_2 + off$, and r_2 points to the stack, then a value-flow constraint is generated to relate the type sets of r_1 and the associated stack slot. Otherwise, no constraints are generated since types of global variables do not change and we do not infer types of heap memory locations.

For a function call, constraints are generated to state that the stack slots are unchanged and values of certain register values are preserved according to the calling convention and the type signature of the callee.

Finally, constraints are also generated based on the control-flow graph. When there is a control-flow edge from instruction $i1$ to instruction $i2$, then the value of a storage location after $i1$ conceptually flows to the storage location before $i2$. The last row in Fig. 5 illustrates the generation of constraints from an example CFG.

In the next step, constraints are decorated with types that are included in debugging information. Fig. 4 shows the syntax of types. Type t can be either common base types, a function type "$(t_1, \ldots, t_n) \rightarrow t$" whose parameter types are t_1 to t_n and return type is t, a pointer type $t*$, a fixed-size array type $t[n]$, or a struct type in the form of a list of fields and field types. We augment C's type system with type tbl_ent, which is used as the type of jump-table entries and helps our system refine targets of indirect jumps. When debugging information tells that the type of x is t, we write $x : t$. For instance, "$\texttt{eax}^{-}_{10}$: int" means that `eax` before address 10 is of type int. We use capital letters X, Y, and Z for *decorated type variables*, which are either x or $x : t$; the second form is used when debugging information tells that x is of type t. A type constraint C is then turned into a *decorated type constraint* D by annotating type variables in C with types provided in debugging information.

4.4 Constraint solving

Decorated type constraints are solved in two steps: (1) constraints are turned into a graph whose nodes are decorated type variables and whose edges model value flows; (2) types are propagated on the graph and a type set for each node is produced.

From constraints to a constraint graph. In the resulting graph, nodes are decorated type variables and edges are of three kinds. For a constraint of the form $X \supseteq Y$, we add a value-flow edge from node Y to node X, written as $Y \rightarrow X$. For a constraint of the form $X \supseteq *(Y + off)$, we add a dereference-flow edge from node Y to X and label the edge with *off*; this is written as $Y \overset{off}{\Rightarrow} X$. For a constraint of the form $X \supseteq \&(Y + off)$, we add an address-flow edge from node Y to X and label the edge with *off*; this is written as $Y \overset{off}{\dashrightarrow} X$. An example is shown in Fig. 6: part (a) shows a set of constraints and part (b) shows its corresponding graph.

Constraint graph solving. Algorithm 1 presents the algorithm for solving a constraint graph. At a high level, it uses a worklist to propagate types forward along edges in the graph and computes a *typeOf* function that assigns sets of types to nodes. In more detail, $typeOf(n)$ is initialized according to n's kind: if n is $x : t$, then its type is t since we trust debugging information; if n has no incoming edges, we initialize its type as $\top$, meaning there is no information about the type; otherwise, it is initialized to be $\emptyset$. Then a while loop is used to process the nodes in the worklist. For a node, the algorithm iterates through its outgoing edges. For a value-flow edge $n \rightarrow n'$, when $n' = y$, the types for n are propagated to n'; note if "$n' = y : t$", then there is no need to perform forward propagation since the type of y is already known. For a dereference-flow edge $n \overset{off}{\Rightarrow} n'$, when $n' = y$, the type is propagated according to each type t in $typeOf(n)$. For an address-flow edge $n \overset{off}{\dashrightarrow} n'$, when $n' = y$, the types for n are transformed into correspondent pointer types and propagated to n'. In the case when t is a struct type, we use $\text{offsetTy}(t, off)$ to compute the field type in the struct according to a static offset for both kinds of edges. Part (c) in Fig. 6 shows the solved example according to the algorithm.

5 CFG CONSTRUCTION

In this section, we present how our system constructs high-precision CFGs for binaries in two stages. The first stage produces a coarse-grained, base CFG. The second stage uses type-inference results to enhance CFG precision.

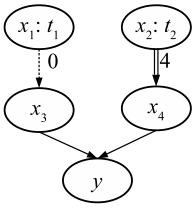
5.1 Base CFG construction

The base CFG construction relies on the classic recursive-traversal disassembly algorithm, which mixes disassembly and CFG construction. It maintains a work list, initialized with known code entries of binaries. The control flows of already disassembled instructions are followed for finding new code addresses to add to the work list for continuing disassembly. It can accommodate data embedded in code since such data should not be targets of control-transfer instructions. However, when a basic block ends with an indirect branch such as an indirect call through a memory operand, without further information it is difficult to predict what the branch may target and add appropriate addresses for further disassembly.

(a) Constraints

$x_3 \supseteq \&(x_1 : t_1)$
$x_4 \supseteq *((x_2 : t_2) + 4)$
$y \supseteq x_3$
$y \supseteq x_4$

(b) Constraint graph



(c) Solved constraint graph

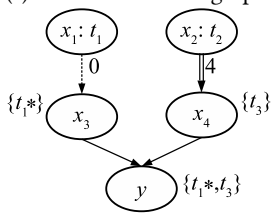


Figure 6: An example for illustrating constraint solving. Assume t_1 = int, t_2 = {f1 : int, f2 : float∗}, and t_3 = float∗.

Algorithm 1 Constraint solving

```
Global:
    worklist : P(N)
    typeOf : N → P(Type)

  procedure SOLVE(G = (N, E))
      worklist ← N
      for n ∈ N do
          switch n do
              case n = "x : t":
                  typeOf(n) ← {t}
              case n = x and n has no incoming edges:
                  typeOf(n) ← {⊤}
              case n = x and n has incoming edges:
                  typeOf(n) ← ∅
      while worklist is not empty do
          n ← remove a node from worklist
          for e ∈ outgoing edges of n do
              switch e do
                  case e = n → y:
                      for t ∈ typeOf(n) do ADD(t, y)
                  case e = n ⇒(off) y:
                      for t ∈ typeOf(n) do
                          switch t do
                              case t = {id_1 : t_1, ..., id_n : t_n}:
                                  ADD(offsetTy(t, off), y)
                              case t = t'* or t'[k]: ADD(t', y)
                              case others: ADD(⊤,y)
                  case e = n -->(off) y:
                      for t ∈ typeOf(n) do
                          switch t do
                              case t = {id_1 : t_1, ..., id_n : t_n}:
                                  ADD(offsetTy(t, off)*, y)
                              case others: ADD(t*, y)

  procedure ADD(t,y)
      if t ∉ typeOf(y) then
          typeOf(y) ← typeOf(y) ∪ {t}
          worklist ← worklist ∪ {y}
```

A tool may ignore the issue by not adding any address, but this would result in incomplete disassembly. Many tools such as IDAPro make assumptions about compilers and rely on heuristics and code patterns for remedying the situation partially, but they still cannot achieve complete disassembly in all cases since it is limited by the amount of information in binary code.

Our implementation of recursive traversal utilizes meta-information generated by compilers to determine control-flow targets of indirect branches and adds those targets to the worklist for further disassembly. Since it retrieves all possible targets for indirect branches from meta-information, it is able to achieve a complete disassembly and a sound base CFG for further refinement in the second stage of CFG construction.

How our recursive traversal algorithm determines the targets of indirect branches depends on the types of indirect branches. For both indirect calls and indirect jumps, we take advantage of relocation information to narrow down potential targets; therefore, we first explain how potential targets for indirect calls/jumps are retrieved from relocation information.

Indirect targets retrieval via relocation information. Since a function cannot be invoked indirectly unless its address is taken somewhere in code, several previous systems [1, 20, 36] refine CFGs to allow indirect calls to target only address-taken functions. This is possible because at compile time the compiler does not know the exact addresses of functions and it must generate relocation entries for places where function addresses are used so that during linking they can be patched when the exact function addresses become known. The same applies to targets of indirect jumps; relocation entries are generated at places where those targets are used.

Therefore, our system searches for relocation entries that require the linker to put in absolute addresses during linking. If the symbol name of such an entry is a function name, the function's address must be taken and it is collected into the set FSA (Function-Start Addresses), which is the set of start addresses of address-taken functions. If the symbol name of such an entry is not a function name, the corresponding address is internal to a function and possibly the target of an indirect jump; the address is then collected into ICA (Internal Code Addresses); that is, it is the set of code addresses that can be the targets of indirect jumps.

Handling indirect calls. In the base CFG, an indirect call is allowed to target any address in FSA, the set of start addresses of address-taken functions; furthermore, since functions contained in dynamically linked libraries are called through entries in the Procedure Linkage Table (PLT) in the binary, our system also adds the start addresses of PLT entries to the targets of indirect calls.

Handling indirect jumps. An indirect jump is allowed to target an address if (1) the address is in FSA, or if (2) the address is in ICA and the address falls within the code-address range of the function that contains the indirect jump. The reason for (1) is that tail calls are

implemented via indirect jumps; an indirect jump for implementing a tail call is like an indirect call and is therefore allowed to target any function whose address is taken. For (2), the justification is that non-tail-call indirect jumps are for compiling switch statements and goto statements through labels that are stored in composite data structures; in these cases, targets of an indirect jump are local (within the function that contains the indirect jump).

Handling returns. When processing a basic block that ends with a direct or indirect call instruction, our recursive-traversal algorithm adds the code address immediately following the call instruction into the disassembly worklist. This is speculative disassembly since it assumes the callee function has a return instruction that will return to the address following the call instruction. The speculative disassembly can produce spurious basic blocks, because the call may target functions that never return; for example, calling the library function `exit` stops the program. Our system assumes that compilers do not put non-executable bytes after call instructions; consequently, spurious basic blocks are not harmful.

With speculative disassembly for return addresses, the targets of return instructions can be computed after the disassembly is complete. Specifically, after computing targets of indirect calls/jumps, our system computes a call graph to compute targets of return instructions. The call graph collects a list of functions that a call instruction can reach, either directly or indirectly through a series of tail calls. Then, return instructions in this list of functions can target the return address following the call instruction.

5.2 Enhanced CFG construction

Base CFG construction results in coarse-grained CFGs: all indirect calls share the same set of targets; indirect jumps are allowed to target either function-start addresses or some local targets or both. In this section, we discuss how to use the results of type inference described in Sec. 4 to substantially enhance the CFG precision. A previous CFI system [20] adopts a type-signature approach for CFG construction: an indirect call through an operand is allowed to invoke any function whose type is compatible with the operand's function pointer's type. This type-based method requires that source code does not contain type casts that involve function-pointer types; if this is violated, a small amount of changes can be made to the source code to respect the requirement.

The type-signature method requires knowing two pieces of knowledge: (1) the types of functions; and (2) the function-pointer types of operands used in indirect calls. As presented before, types of functions can be acquired from debugging information. However, debugging information does not always tell the types of operands in indirect calls; this is where our type inference comes into play. After type inference, an operand in an indirect call is assigned a set of types. For each function-pointer type $(t_1, \ldots, t_n \rightarrow t)*$ in the set, we allow an indirect call via the operand to invoke any function with type "$t_1, \ldots, t_n \rightarrow t$". In the worst case, the set may contain $\top$; for example, when the operand is loaded from a piece of memory for which there is no type information; in this case, the target-set of the indirect call cannot be refined.

For an indirect jump, thanks to the new type tbl_ent, the type-inference also tells whether it is a tail call or a local jump based on jump tables. If the type set of the indirect jump's operand contains only function-pointer types, then it is treated as a tail call and its target set is refined as if it were an indirect call. If the type set of the operand contains only tbl_ent, the set of targets is refined to include only the local targets. Each PLT entry also contains an indirect jump; we allow it to target the address of the dynamic linker as well as a symbolic target that indicates that the jump is allowed to jump into a dynamically linked library. The target sets of remaining indirect jumps are not refined.

After the target sets of indirect calls and jumps are refined, a refined call graph is constructed. The refined call graph is then used to calculate the targets of return instructions.

6 IMPLEMENTATION AND EVALUATION

Our system is implemented with a decoder for x86-32, 18K lines of OCaml code for disassembly, type inference, and CFG construction, 4K lines of C code for collecting debugging information, and a few Python and Shell scripts for relocation information collection and data handling.

The disassembler is built based on the decoder. The disassembler first parses the ELF file to obtain all headers and sections. Symbol tables are already retrieved during ELF parsing. Relocation information is collected during compilation with the help of the Linux utility `readelf` and a custom Python script. Debugging information is collected with the help of the `libdwarf` library, implemented in C. Disassembly and CFG construction are implemented in OCaml and communicate with the debugging-information collection in C via the OCaml-C interface.

For evaluation, we are interested in the following questions: (1) how effective is our type inference for inferring types of operands in indirect branches? (2) How precise are the CFGs generated by our system and how does the precision compare with the binary-analysis approach and the compiler-modification approach?

To answer these questions, we conducted our experiments in a Linux box running x64 Ubuntu 14.0.4 on a PC with 16GB-memory and Intel Core i5-4590 CPU at 3.30GHz. Our evaluation was performed on SPEC2006 C benchmarks and Nginx-1.4.0. Since the type-based approach requires that source code does not contain type casts that involve function-pointer types, we used MCFI's patched SPEC2006 C benchmarks and Nginx-1.4.0 (downloaded from https://github.com/mcfi); MCFI made small changes to the benchmarks to remove type casts that involve function pointers (mostly by adding function wrappers). Our prototype system can construct CFGs for both GCC and LLVM at all optimization levels (from `O0` to `O3`). We also tested the CFG construction on multiple versions of the two compilers (in particular, GCC 4.8.4, GCC 5.4.0, GCC 6.2.0, LLVM 3.9, LLVM 4.0). For conciseness, we will present the detailed results only for GCC 4.8.4 and results are generally similar for other GCC versions and LLVM.

6.1 Effectiveness of type inference

Tbl. 1 presents indirect-branch type-inference results for SPEC2006 C benchmarks. In the table, the benchmarks are sorted according to their sizes in the ascending order. We omitted small benchmarks including `lbm`, `mcf` and `libquantum` since they do not contain indirect calls/jumps. For each benchmark, column NeedInfer/Total lists the number of indirect calls whose operands do not have debugging type information versus the total number of indirect calls. The

Benchmarks	NeedInfer/ Total	ICALL-Precise	ICALL-Imprecise	Inferred Rate	TailCall	Local	PLT	IJUMP-Imprecise
bzip2	20/20	20	0	100.0%	0	2	22	0
sjeng	1/1	1	0	100.0%	0	15	38	0
milc	0/4	0	0	N.A.	0	5	40	0
sphinx3	9/9	9	0	100.0%	0	1	60	0
hmmer	0/9	0	0	N.A.	1	24	69	0
h264ref	40/352	40	0	100.0%	0	12	44	0
perlbench	55/109	52	3	94.5%	30	80	114	0
gobmk	30/44	30	0	100.0%	0	13	49	0
gcc	292/442	291	1	99.7%	20	426	72	1
Total	447/990	443	4	99.1%	51	578	508	1

Table 1: Indirect-call and Indirect-jump type-inference results (GCC 4.8.4 with O2 optimization).

numbers show that there are many indirect calls (around 50% for large benchmarks including perlbench, gobmk, and gcc) for which debugging information misses types for their operands.

Column ICALL-Precise shows the number of indirect calls whose operands type inference infers a single function-pointer type for; Column ICALL-Imprecise shows the number of indirect calls when type inference includes $\top$ or non-function-pointer types in the inferred type set; Column Inferred Rate shows the percentage of indirect calls for which type inference can give precise function-pointer types (column ICALL-Precise) over the indirect calls whose operands debugging information carries no types for. The numbers show that type inference can infer the types for a high percentage (over 99%) of indirect calls.

We also investigated those imprecise cases. Some were caused by union types. For instance, one call site in perlbench has a union type for the operand. Since the inference system does not know which case of the union type is the operand's exact type, it collects all cases from the union type for the operand's type set; and one of those is void*, a non-function-pointer type. Other cases were caused by over-approximation in our constraint generation since it does not track the types of storage locations in the heap.

The second half of Tbl. 1 presents indirect-jump type-inference results. Column TailCall presents the number of indirect jumps our system infers as tail calls (recall that the operand type of such an indirect jump should be a function-pointer type). Column Local presents the number of indirect jumps that are inferred as jumps via jump tables. A large number of indirect jumps are in PLT entries and their numbers are shown in Column PLT. Column IJUMP-Imprecise shows the number of indirect jumps whose operand types include $\top$. The data shows that there is only one indirect jump for which our system cannot infer precise information.

6.2 CFG precision and validation

For a control-flow graph, Average Indirect-Branch Targets (AIBT) introduced by Niu [18] is defined to be the number of targets averaged over all indirect branches in the graph.[3] The smaller the AIBT is, the more precise the CFG is. In Tbl. 2, we present AIBT numbers for SPEC2006 C benchmarks for GCC 4.8.4 across different optimization levels. Each data entry shows AIBTs for the enhanced CFG (left) and the base CFG (right).

Comparison with binary-analysis approach. The precision of base CFGs is roughly the same as the precision of CFGs produced by a typical binary-analysis approach, which commonly allows all indirect calls to target the same set of functions and employs optimizations for handling indirect jumps. As the table shows, the improvement of our type-inference based CFG enhancement is small for small benchmarks; since lbm, mcf, and libquantum do not contain indirect branches, there is no improvement during CFG enhancement. However, for large benchmarks including perlbench, gobmk, and gcc, the CFG improvement is substantial: average reduction is between 60 to 90%; many spurious edges were removed during type-based CFG enhancement.

Comparison with compilter-modification approach. When compared with the compiler-modification approach, the precision of our system's enhanced CFGs is only slightly worse than MCFI. A direct comparison by the AIBT statistics is infeasible, because MCFI's implementation does not support 32-bit binaries and our current implementation supports only 32-bit binaries. However, based on the fact that both systems use the type-signature approach for matching indirect branches and targets, we can still perform an indirect comparison. MCFI propagates types inside the compiler to binaries; so an indirect call's operand has exactly one type. Our system performs type inference to infer the types of indirect-call operands. As seen in a previous table, around 99% of indirect-call operands get one type signature. Therefore, the CFG precision of our system is very close to the one of MCFI. As a previous survey paper [5] shows, MCFI enforces the highest-precision CFGs among all existing CFI enforcement tools. For instance, Forward-Edge CFI [27] enforces a less precise CFG: it uses arity matching for pairing indirect calls and functions (i.e., it matches the number of parameters) and it does not enforce CFI on return instructions.

Experiment on Nginx 1.4.0. To further evaluate our approach, we conducted an experiment on Nginx. In Tbl. 3, we present experimental results for MCFI-patched Nginx-1.4.0. As the table shows, our approach is able to generate high-precision CFGs for practical applications such as Nginx.

[3] We did not use the AIR (Average Indirect-target Reduction) metric because it has been criticized to not capture the ability of a CFG to withstand control-flow hijacking attacks [7].

Benchmarks	O0	O1	O2	O3	AvgRed
lbm	1.7/1.7	1.7/1.7	1.7/1.7	1.7/1.7	0%
mcf	1.6/1.6	1.6/1.6	1.5/1.5	1.4/1.4	0%
libquantum	3.0/3.0	3.2/3.2	4.0/4.0	4.8/4.8	0%
bzip2	2.6/3.0	2.7/3.0	2.5/2.7	1.9/2.2	11.1%
sjeng	5.0/9.0	4.9/4.9	6.5/6.5	6.8/6.8	11.1%
milc	3.3/3.3	3.3/3.3	3.6/3.6	3.9/3.9	0%
sphinx3	4.5/4.6	4.7/4.7	4.9/4.9	5.7/5.7	0.5%
hmmer	4.1/7.8	4.8/4.9	4.9/5.0	4.9/4.9	12.9%
h264ref	4.0/31.8	4.3/32.1	6.1/34.7	6.2/34.9	84.7%
perlbench	17.2/563.0	19.5/102.8	23.6/232.6	34.1/357.6	89.6%
gobmk	19.7/104.6	22.0/61.2	19.5/58.1	28.9/68.7	67.4%
gcc	23.2/1735.0	26.2/192.8	32.3/212.3	42.3/345.5	89.4%

Table 2: Average Indirect-Branch Targets for SPEC benchmarks (GCC 4.8.4).

Benchmarks	NeedInfer/ Total	ICALL- Precise	ICALL- Imprecise	Inferred Rate	TailCall	Local	PLT	IJUMP- Imprecise	AIBT	Red
nginx	201/289	191	10	95.0%	26	60	133	0	20.7/213.0	90.3%

Table 3: Experimental results for Nginx-1.4.0 (GCC 4.8.4 with O2 optimization).

CFG validation. We have performed testing to validate the enhanced CFGs our system generated for SPEC benchmarks and Nginx. We used PIN [15] to instrument the benchmarks' binaries to collect runtime traces for reference data sets and configurations; the reference data sets are included in the original benchmarks and provide good test coverage. The control-flow edges made by indirect branches in the runtime traces were checked to see if they were included in the CFGs generated by our system. In our experiments, CFGs generated by our system for all benchmarks passed the validation. Because the runtime overhead introduced by PIN is substantial and the trace output file is large, we had to perform optimizations for large benchmarks during validation. Since the difficulty of CFG construction lies in control transfers from indirect branches, we instrumented only indirect branches through PIN instead of all control-flow transfer instructions. One downside of resorting to testing for CFG validation is that it can find bugs but cannot show correctness. Unfortunately, the CFI field lacks a method for verifying the correctness of CFGs; addressing this problem would be an interesting research direction.

7 RELATED WORK

Previous binary-level type-inference systems assume stripped binaries (Caballero and Lin [6] provide a comprehensive survey). In contrast, our system uses debugging information for type inference. This comes with the downside of assuming source-code availability, but it enables more complete type inference since debugging information provides a wealth of source-level types. Thanks to debugging information, our flow-based type inference can infer most of the types in operands of indirect branches, while previous systems on stripped binaries had trouble of inferring function-pointer types [6], even after employing more complex techniques including heuristics (e.g., for classifying whether an immediate is an integer or a pointer), points-to analysis, and dynamic analysis.

Static disassembly of binaries and CFG construction have always been a challenge in reverse engineering binary programs. Other than the standard linear-sweep and recursive-traversal algorithms, other attempts at better static disassembly and CFG construction have been presented in previous papers. Kruegel *et al.* [14] proposed to combine linear sweep with recursive traversal. The system does not follow the control flows of indirect branches and suffers from incomplete disassembly. Balakrishnan *et al.* proposed Value Set Analysis (VSA [3]) that can sometimes statically determine the control-flow targets of indirect branches. Wartell *et al.* [32, 33] described a machine-learning approach for discovering the most likely disassemblies with the help of a training corpus. TypeArmor [28] performs liveness-analysis based arity matching to refine the control-flow targets of indirect calls. All previous systems assume stripped binaries.

8 CONCLUSIONS

We have designed and implemented an alternative approach for high-precision CFG construction, without compiler modification. The approach uses compiler-generated meta-information to retrieve source-level information for CFG construction. It relies on a type-inference engine that deduces types of indirect-branch operands from source-level types in debugging information. Our experiments demonstrate that the precision of CFGs produced by our system is comparable to previous systems that modify compilers and our system is compatible with multiple compilers and multiple compiler versions, thanks to its compiler-independent design. As future work, we plan to expand the implementation to cover C++ applications by adding the support for classes.

Acknowledgments. We thank reviewers for their insightful comments. This research is based upon work supported by US NSF grants CCF-1624124 and CNS-1624126. The research was also supported in part by the Defense Advanced Research Projects Agency (DARPA) under agreement number N6600117C4052 and Office of Naval Research under agreement number N00014-17-1-2539.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th Usenix Security Symposium*. 583–600.

[3] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *13th International Conference on Compiler Construction (CC)*. 5–23.

[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification (CAV)*. 463–469.

[5] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1 (2017), 16:1–16:33.

[6] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *Comput. Surveys* 48, 4 (2016), 65:1–65:35.

[7] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th Usenix Security Symposium*. 161–176.

[8] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd Usenix Security Symposium*. 385–399.

[9] Mihai Christodorescu and Somesh Jha. 2003. Static Analysis of Executables to Detect Malicious Patterns. In *12th Usenix Security Symposium*. 169–186.

[10] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd Usenix Security Symposium*. 401–416.

[11] DWARF Debugging Information Format Committee 2017. *DWARF Debugging Information Format Version 5*. DWARF Debugging Information Format Committee.

[12] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 575–589.

[13] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security Symposium (NDSS)*.

[14] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *13th Usenix Security Symposium*. 255–270.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

[16] Julia Menapace, Jim Kingdon, and David MacKenzie. 1999. *The "stabs" debug format.*

[17] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 395–404.

[18] Ben Niu. 2015. *Practical Control-Flow Integrity.* Ph.D. Dissertation. Lehigh University, Bethlehem, PA.

[19] Ben Niu and Gang Tan. 2013. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *20th ACM Conference on Computer and Communications Security (CCS)*.

[20] Ben Niu and Gang Tan. 2014. Modular Control Flow Integrity. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 577–587.

[21] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *21st ACM Conference on Computer and Communications Security (CCS)*. 1317–1328.

[22] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering class Hierarchies in C++ Programs. In *Network and Distributed System Security Symposium (NDSS)*.

[23] Jannik Pewny and Thorsten Holz. 2013. Control-Flow Restrictor: Compiler-based CFI for iOS. In *ACSAC '13: Proceedings of the 2013 Annual Computer Security Applications Conference*.

[24] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. 2010. There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code. In *Computer Aided Verification (CAV)*. 41–56.

[25] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*. 552–561.

[26] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security.*

[27] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd Usenix Security Symposium*.

[28] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE Symposium on Security and Privacy (S&P)*. 934–953.

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. 1993. Efficient Software-Based Fault Isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM Press, New York, 203–216.

[30] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 380–395.

[31] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 299–308.

[32] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. 2014. Shingled Graph Disassembly: Finding the Undecidable Path. In *Proceedings of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. Tainan, Taiwan, 273–285.

[33] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, Vol. 3. 522–536.

[34] Zhichen Xu, Barton Miller, and Thomas Reps. 2000. Safety checking of machine code. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 70–82.

[35] Bennet Yee, David Sehr, Gregory Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (S&P)*.

[36] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (S&P)*. 559–573.

[37] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd Usenix Security Symposium*. 337–352.