FPValidator: Validating Type Equivalence of Function Pointers On The Fly

Hua Wang *, Yao Guo * and Xiangqun Chen *
* *Key Laboratory of High Confidence Software Technologies (Ministry of Education),*
Institute of Software, School of EECS, Peking University
{*wanghua04, yaoguo, cherry*}*@sei.pku.edu.cn*

***Abstract*—Validating function pointers dynamically is very useful for intrusion detection since many runtime attacks exploit function pointer vulnerabilities. Most current solutions tackle this problem through checking whether function pointers target the addresses within the code segment or, more strictly, valid function entries. However, they cannot detect *function entry attacks* that manipulate function pointers to target valid function entries but invoke them maliciously.**

This paper proposes FPValidator, a new solution capable of dynamically validating the type equivalence between function pointers and target functions, which can detect all function entry attacks that violate type equivalence. An effective and efficient type matching approach based on *labeled type signature* is proposed to perform fast type equivalence checking. The validation code and necessary type information are inserted by a compilation-stage instrumentation mechanism, bringing no extra burden to developers. We integrate FPValidator into GCC and evaluation shows that its performance overhead is only about 2%.

I. Introduction

Adversaries often intend to convert programs' control flow, so that they can take programs under their control. For programs written in languages that support function pointers, manipulating function pointers is an often-used attacking method. An adversary could modify a function pointer to target an illegal address in order to serve his malicious purpose. A well-known attack to tamper with function pointers is to exploit buffer overflow vulnerabilities [1]. Adversaries could also modify function pointers from outside, for example by using *procfs*. Since function pointers may be maliciously changed during execution, validating their values on the fly is clearly useful for intrusion detection.

A key problem with dynamic function pointer validation is how to judge that a pointer's value is a valid address. We need to define for each pointer a set of addresses that the pointer is allowed to point to during execution. Ideally, a set should satisfy the following two requirements, used to avoid false-positives and false-negatives separately.

- *Completeness*: The set should contain *all* addresses that the pointer probably targets in correct execution.
- *Precision*: The set should *only* contain the addresses that the pointer probably targets in correct execution.

To some extent the two requirements are conflicting. Perfectly meeting both of them is very hard. It is usually considered that completeness is prior to precision in that users do not like to be interrupted by fake errors [2]. Existing solutions, after guaranteeing completeness, try to improve precision as far as possible. Several solutions define the set as all addresses in code segments, or more strictly, valid function entries [3], [4]. Although these solutions can meet completeness well, they are not precise enough to detect *function entry attacks*. Function entry attacks mean that an adversary manipulates a function pointer to target a valid function which however should not be invoked by that pointer. This kind of attacks could cause serious security problems. For example, an adversary could modify a pointer to invoke the *system* function, which is contained in *libc.so* and linked into most programs' address spaces, to execute an arbitrary command.

Some solutions try to find out accurate points-to sets. Inlined CFI [5] deduces the sets from a CFG obtained by static analysis. Unfortunately, such a CFG is usually not accurate due to indirect branches. WIT [6] uses the static points-to analysis [7] to compute the sets, but the result can hardly be precise [8]. In the worst case, the set for a pointer could contain all function entries.

In this paper we propose a new solution called *FPValidator*, which uses a new method of defining the set of possible targets, being precise enough to effectively detect function entry attacks, as well as having no false-positives. Our method is based on the fact that, in a statically-typed language (e.g. C and C++), a function pointer should only invoke functions of compatible types; otherwise the program's behavior could be unpredictable. Consequently, we define the set for a function pointer as all functions whose types are compatible with the pointer. With our method, an adversary, if succeeding in tampering with a function pointer, can only abuse compatible functions, making function entry attacks much harder.

FPValidator validates function pointers through a fast type matching method. In terms of programs developed with statically-typed languages, efficiency is often an important factor that needs to be considered. We should therefore minimize the cost caused by dynamic type matching. In

This work has been supported by the National High Technology Research and Development Program (863) of China under Grant No. 2007AA010304, 2007AA01Z462 and 2008AA01Z133, the National Basic Research Program of China (973) under Grant No. 2009CB320703, and the Science Fund for Creative Research Groups of China under Grant No. 60821003.

fact, we do not need to adopt a complete and complicated dynamic type system like those for dynamically-typed or hybrid languages (e.g. [9], [10], [11], [12], [13]) which usually causes considerable runtime overhead. Instead, we propose a lightweight, fast type matching method based on *labeled type signature*. Our method calculates the hashes of type signatures of function pointers and functions during compilation, and compares them during validation. Type information and validation code fragment are inserted into programs automatically by a compilation-stage instrumentation technique, causing no extra burden to developers. The evaluation shows that the increased time cost is only about 2%, and the increased space cost is less than 8%.

The rest of this paper is organized as follows. Section 2 analyzes function pointer attacks, and demonstrates an example of function entry attack. Section 3 describes the details of our validation method, including the type matching and the instrumentation. Section 4 describes the implementation and evaluation, as well as the security analysis. The related work is discussed in Section 5. Finally we conclude this paper in Section 6.

II. Function Pointer Attacks

When attacking function pointers, adversaries usually modify function pointers to point to the code that could serve their malicious purposes. Depending on where the modified function pointers target, we divide function pointer attacks into *data region attacks (DRA)*, which target the code injected into data segments, and *code region attacks (CRA)*, which target the addresses within code segments. CRA could be further divided into *function entry attack (FEA)* and *non function entry attack (NFEA)* (also known as arc injection attack [1], [14]), according to whether the function pointers target function entries.

On the other hand, function pointer validation, according to precision, can be divided into four levels [4], listed as follows.

- L1: the target must be in code segments.
- L2: the target must be a predefined position such as a function entry.
- L3: the type of the function pointer and that of the target function must match.
- L4: the target must belong to an accurate points-to address set.

As the level increases, the validation becomes more precise and can detect more attacks. The L1 can only detect DRA, while the L2 can detect both DRA and NFEA. The L3, stepping further, can detect all FEA that involves type violation, and the L4 can detect almost all function pointer attacks.

FEA could cause serious security problems. Programs often contain sensitive functions which should be called at right time and with right parameters. Besides, some functions, although contained in code segments, should never

```
1  typedef void (*func_t)(char*, int);
2
3  void foo(char *name, int type) {
4      ...
5  }
6
7  void func(char* inbuf, int len) {
8      func_t fp;
9      int param;
10     char buf[16];
11     ...
12     fp = foo;
13     param = IPV4;
14     memcpy(buf, inbuf, len);
15     fp(buf, param);
16     ...
17 }
```

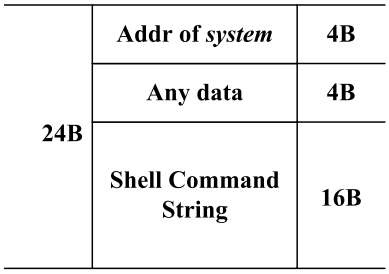


Figure 1. A buggy function and malicious input

be called in normal execution. For example, shared libraries (e.g. *libc.so*) are linked as a whole into programs' address spaces, but many functions they contain are never used. Some of these functions are "dangerous", such as *system* and *execve*, which may be invoked to execute arbitrary commands. These functions may be abused by adversaries through FEA.

Figure 1 shows a buggy function that could be exploited by FEA. Line 8 declares a local function pointer, *fp*, followed by a declaration of an local variable. Both may be overwritten by the buffer operation of line 14. Normally *fp* will point to *foo*, and at line 15 *foo* gets to run. An adversary, however, could deliberately construct a "bad" input to change *fp*, calling another function instead. For example, assuming the program is running on a 32-bit platform, an adversary can construct an input buffer as in Figure 1. The size of the input buffer is 24 bytes, containing at most 16 bytes of a shell command string and the address of the *system* function. The command string will be copied to *buf*, and *fp* will be modified to point to *system*. Later when *system* is called through *fp*, the adversary's command is executed. In this example, because the new target of *fp* is a legal function entry, the attack can not be detected by the first two validation levels.

In addition to the buffer overflow vulnerability, adversaries could also use other attacking methods to tamper with function pointers, such as writing memory directly through

procfs. As long as they only exploit "legal" functions, their attacks can not be detected by the L1 and L2 validation.

III. Validating Type Equivalence

To mitigate the threat of FEA, FPValidator confines function pointers to targeting only compatible functions. To enforce this constraint, FPValidator compares at runtime the type that a function pointer should point to and the type it actually targets. Hereafter the former type is called *point-to type* and the latter is called *target type*. If the two types do not match, an exception is raised. With FPValidator, for an adversary, only when the point-to type matches the type of the function he wants to abuse does he have chances to carry out a successful attack, making attacking much harder.

For statically-typed languages, this constraint is safe, that is, it does not cause false-positives. When using statically-typed languages, a developer exactly knows what type of functions are desired for each indirect call statement. Invoking a function of incompatible type is obviously an error. It is notable that sometimes a developer could perform type casting on a pointer in order to invoke functions of other types. In this case the type after casting is treated as the point-to type of this pointer. Consequently, type casting has no impact on the safety of the constraint.

We propose a fast type matching approach to satisfy the efficiency requirement of critical programs. This section describes the design of our approach, as well as the supporting compilation-stage instrumentation mechanism. We use the C language as the example, but our solution can also be applied to other statically-typed languages.

A. Type Matching

In order to detect type violations, we need to define criteria to determine whether the point-to type of a function pointer matches its target type. Considering that function pointers are usually used to invoke functions of exactly identical types, and compilers will generate warning information if their types are different, we require that the point-to type and target type must be equivalent. In general there are two notions of type equivalence, namely structural equivalence and name equivalence [15], but both of them have weak points for function pointer validation.

Intuitively, structural equivalence suggests that the expressions of two types are structurally identical. We can express it in a more formal way. Types, except for basic types, are composed by type constructors using other types. In fact basic types can also be regarded as being composed by special constructors using no other types. The composing relationship can be expressed as

$$\tau = \mathcal{T}(\tau_1, \tau_2, ..., \tau_n)$$

where $\mathcal{T}$ is a type constructor, such as *function* and *record*, and τ_1 to τ_n are depended types. Then $\tau = \mathcal{T}(\tau_1, \tau_2, ..., \tau_n)$ and $\tau' = \mathcal{T}'(\tau'_1, \tau'_2, ..., \tau'_n)$ are structurally equivalent if $\mathcal{T} = \mathcal{T}'$, and $\forall 1 < i < n$, $\tau_i = \tau'_i$. Structural equivalence does not consider type names and field names of *record* types, which usually carry application-specific semantic information and should be used to distinguish types. From programmers' perspective, in most cases two types with different names are not treated as being equivalent even though the structures of their expressions are completely equal, so do two record types whose field names are not identical.

Name equivalence means that two named types are equivalent if their names are identical. In practice, however, it is possible that two types are different but they have the same name. Type names are unique within a compilation unit, but not across units. For programs with multiple source files, because source files are usually compiled separately, it is possible and allowed that different types with the same name are declared in different files if their scopes do not interfere with each other. Thus name equivalence can not distinguish these types.

To overcome the drawbacks of the two notions, we propose *labeled structural equivalence*, a mixture of structural equivalence and name equivalence. First we modify type constructors by adding labels to each depended types. The composing relationship is then expressed as

$$\tau = \mathcal{T}(< \tau_1, l_1 >, < \tau_2, l_2 >, ..., < \tau_n, l_n >)$$

where l_i is the label for the τ_i that carries application-specific information of τ_i. For some constructors the labels are fixed, such as *function* which is expressed as $function(< \tau_1, ret >, < \tau_2, p_1 >, ..., < \tau_n, p_n >)$; while for others they depend on how the type is defined, such as *record* which is expressed as $record(< \tau_1, f_1 >, ..., < \tau_n, f_n >)$ where f_i is the name of the *i*th field.

Based on the modified type constructors we can establish a labeled type graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. $\mathbf{V}$ is the set of vertices, each of which is a pair $< \mathcal{T}, \mathcal{N}(\tau) >$ standing for a type τ built by the constructor $\mathcal{T}$ with the name $\mathcal{N}(\tau)$. For anonymous types, $\mathcal{N}(\tau)$ is null. $\mathbf{E}$ is the set of directed edges denoting the building relationship. For a constructor $\mathcal{T}$, the edge connecting τ and τ_i is labeled with l_i. Figure 2 shows a simple list interface and its labeled type graph.

To simplify the type matching, type qualifiers, e.g. *const* and *volatile*, are not taken into account, and we also do not distinguish signed and unsigned types, because this kind of type violations can hardly be exploited. We also do not consider type names declared by **typedef** statements. They are simply treated as aliases and replaced by its original type in the type graph, because in practice typedefed types and their origins are usually interchangeable.

For a type τ, we define its *definition graph* $\mathcal{D}(\tau)$ as the subgraph containing all reachable nodes from τ and edges connecting them, which completely defines τ. For example the definition graph of *list* is shown in Figure 3. Then we define the labeled structural equivalence of type τ and τ'

```
1  struct list {
2      struct list *next;
3      void *value;
4  };
5
6  typedef int (*walker)(void*);
7
8  void walk_list(struct list *lst, walker w) {
9      ...
10 }
```

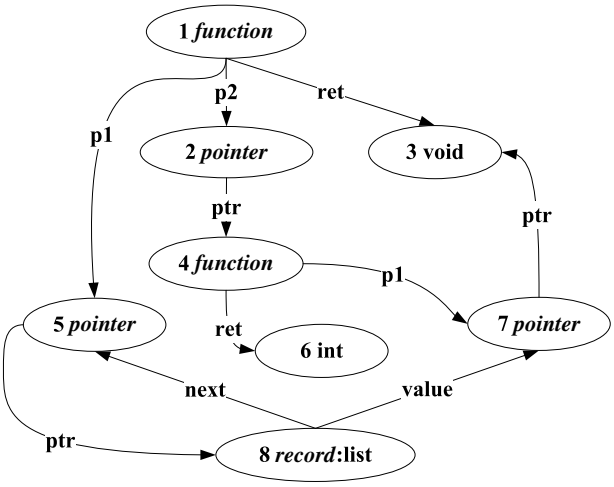


Figure 2. Example code and its labeled type graph

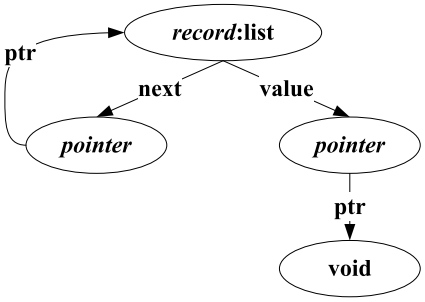


Figure 3. Definition graph of list

as that $\mathcal{D}(\tau)$ and $\mathcal{D}(\tau')$ are completely identical, including their structures, type names and edge labels. More formally, type $\tau = \mathcal{T}(< \tau_1, l_1 >, ..., < \tau_n, l_n >)$ and $\tau' = \mathcal{T}'(< \tau'_1, l'_1 >, ..., < \tau'_n, l'_n >)$ is labeled structural equivalent if $\mathcal{N}(\tau) = \mathcal{N}(\tau')$, $\mathcal{T} = \mathcal{T}'$, and $\forall 1 < i < n$, $\tau_i = \tau'_i$ and $l_i = l'_i$.

Labeled structural equivalence concerns not only type information but also programmer-defined semantic information, being more precise than simply structural equivalence or name equivalence. It can overcome the deficiencies mentioned above.

B. Representing and Storing Type Information

In order to validate types of function pointers, we need to store sufficient type information of function pointers and target functions. On the other hand, to satisfy the efficiency requirement, we expect that this information brings only a little storage overhead and the type matching based on this information is effective and efficient.

As mentioned previously, a type is completely defined by its definition graph, so, for a function pointer or a function, we only need to store the definition graph of its type. Because the definition graph may be compared across compilation units, we should develop a representation scheme which can ensure that the same definition graph appearing in different units is represented in the same form. To do this, we propose a scheme capable of converting a definition graph to a string, referred to as *labeled type signature*, which is similar to but more precise than the traditional type signature [16], [17], [18].

We use $\mathcal{S}(\tau)$ to denote the labeled type signature of type τ. Then, for $\tau = \mathcal{T}(< \tau_1, l_1 >, < \tau_2, l_2 >, ..., < \tau_n, l_n >)$, $\mathcal{S}(\tau)$ is recursively defined as:

$$\mathcal{S}(\tau) = \mathcal{T} \ \ \mathcal{N}(\tau) \ \ l_1(\mathcal{S}(\tau_1))l_2(\mathcal{S}(\tau_2))...l_n(\mathcal{S}(\tau_n))$$

At the beginning of $\mathcal{S}(\tau)$ is the type constructor, followed by the type name, and then the label and recursive labeled type signature of each depended type. The recursion terminates when τ is a primitive type, because it depends on no other types. For example the function pointer type *walker*, which is declared by a **typedef** statement and denoted by node 2 in the labeled type graph, has the following labeled type signature.

$$\mathcal{S}(walker) = \\ pointer\ ptr(function\ ret(int)p1(pointer\ ptr(void)))$$

Some types may be defined recursively, such as *link* which depends on a pointer type that points to itself. To prevent infinite loops in this case, we adopt an alias approach to eliminate recursive dependence. While we are traversing the definition graph, a type is assigned an unique alias at the first time it is met. Later when the type is met again, we simply use its alias to represent it, without recursively traversing its depended types. We use the sequence in which types are traversed as type aliases. For instance $\mathcal{S}(list)$ is equal to

$$record\ list\ next(pointer\ ptr(T1))value(pointer\ ptr(void))$$

where $T1$ is the alias of $list$.

The relationship between the type definition graph and the labeled type signature is 1 to 1. That is to say, throughout the whole program including all compilation units, each type has a unique signature, and, on the other hand, the definition

graph can be accurately rebuilt according to the signature. Hence we could use labeled type signatures to precisely validate the type equivalence between function pointers and their target functions.

Different labeled type signatures usually have different length. Some signatures could be very long. In order to simplify storing and comparing labeled type signatures, we hash signatures and only store and compare their hash values. The signature hashes for functions are stored in a separate section of ELF files (executables or dynamic libraries). When an ELF file is loaded, these signature hashes are read in and organized as a hash table, indexed by function entries. For executables, the addresses of function entries are determined at the linking stage, so they can be stored along with signature hashes. However, for dynamic libraries, these addresses are determined at the load stage, so they are calculated after the load point of the library is determined.

The hash table is marked as read-only after signature hashes are hashed in, in order to prevent malicious modifications. The signature hashes for function pointers are treated as constants stored in read-only data section, which is discussed in details in the next subsection.

C. Instrumenting Programs

FPValidator checks function pointers at runtime. In order to prevent adversaries from circumventing the dynamic validation, it is desired that the validation is performed immediately before each indirect call. We use a compilation-stage instrumentation mechanism to achieve this goal.

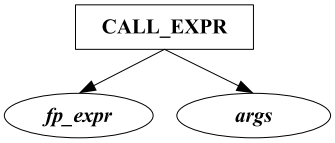
First of all, we need to identify indirect function calls. In general a call can be expressed as $(fp\_expr)(args)$, where fp_expr could be an address, a function pointer variable or a complex expression consisting of operations such as array indexing, field references, type castings, or even another call, and $args$ is the parameter list. Whatever fp_expr is, as long as its value may change at runtime, the call is treated as an indirect call and the type of target functions should be the type of fp_expr. To determine whether fp_expr could change, we traverse its Abstract Syntax Tree (AST). If its value directly or indirectly depends on variables or the results of function calls, the call is treated as an indirect call and will be instrumented. In fact, if the value of a complex expression may change, the expression as a whole can be viewed as an implicit function pointer variable. FPValidator validates not only function pointer variables but also function pointer expressions. It has wider coverage than solutions that only validate function pointer variables, such as [4].

Each indirect call is translated, by instrumentation, into the code fragment shown in Figure 4. If fp_expr is a complex expression, it may have side effect. To prevent it from being executed multiple times, at the beginning we save the target address it generates into a local variable $\_\_fp$, and later use the variable instead. *__HASH__* stands for the hash of fp_expr's labeled type signature, which is

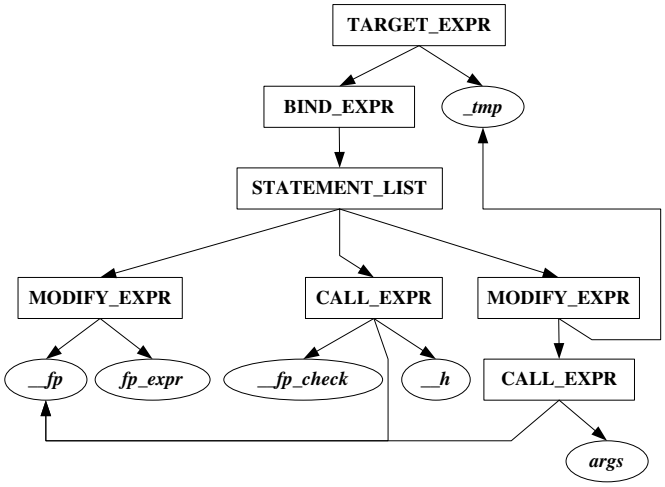
```
({
    typeof(fp_expr) __fp = fp_expr;
    hash_t __h = __HASH__;

    __fp_validate(__fp, __h);
    __fp(args);
})
```

Figure 4. Code after instrumentation



(a) Original AST



(b) AST After Instrumentation

Figure 5. Instrumenting AST

generated at compilation time and declared as a constant. Its location is therefore within the read-only data section. During execution, *__HASH__* is assigned to a local variable, which is in turn passed as a parameter to $\_\_fp\_validate$, the validating function, along with the value of $\_\_fp$ (i.e. the target address). We enclose all statements as a compound statement whose value is equal to that of its last statement, i.e. the result of the indirect call, in that the result may be used as a right value in some cases, such as the statement $a = (fp\_expr)(args)$.

$\_\_fp\_validate$ is the function containing the actual validation code, which is implemented in a shared object and linked dynamically, so that updating the validation code is convenient, without requiring to recompile the program. $\_\_fp\_validate$ looks up the hash of the type signature of the target function in the hash table according to $\_\_fp$, the target address, and then compares it with $\_\_h$. If they are identical, the function pointer and the target function are type equivalent, otherwise the indirect call is illegal.

We accomplish the instrumentation through transforming AST. AST is generated by the compiler, which contains sufficient type information that we can use. We traverse the AST of each function, finding nodes that represent indirect call expressions, and replacing each of them with a new AST corresponding to the code segment shown in Figure 4. The original AST of an indirect call expression is depicted in Figure 5(a). After instrumentation, according to whether the return type of the indirect call is **void**, the original AST is replaced by a *BIND_EXPR* or a *TARGET_EXPR* separately. The AST for these two expressions are similar, except for some subtle difference. We only show the AST of the latter in Figure 5(b).

IV. Implementation and Evaluation

FPValidator can be applied to programs written in statically typed languages. We have integrated FPValidator into *gcc* for GNU C. This section describes our implementation and its evaluation.

A. Implementation

Our implementation includes two parts, namely an instrumenting module (IMod) and an ELF rewriter (ELFRwt). IMod is responsible for collecting type information and instrumenting indirect call statements, while ELFRwt is responsible for rewriting the generated ELF file to add the type hashes of functions. The architecture of our implementation is shown in Figure 6.

IMod inserts the validation code at the compilation stage, so we need to modify the compilation process. To minimize the modification, we build IMod upon GCC Extension Modules (GEM) [19]. GEM patches *gcc*, adding a group of hooks which can be used to invoke user-defined functions at certain points. User-defined functions are encapsulated in dynamically linked libraries which can be specified by command-line options. IMod is implemented as a GEM module, defining the function for the hook *gem_finish_function* which is called after a function has been parsed and its AST has been generated. In the hook function we traverse the AST, searching for indirect call expressions and instrumenting each of them. The instrumented AST is used later to generate the ELF file. IMod also collects the type information of functions, i.e. the hashes of type signatures, which is passed to ELFRwt for building the type hash table.

The final ELF file is built by ELFRwt. ELFRwt is implemented as a separate program executed after the ELF file has been generated by *gcc*. It rewrites the ELF file, adding a dedicated section to store the signature hashes of functions. These hashes could be obtained from IMode if the source code is available. However, sometimes we can not get the source code, for example, when dealing with a dynamic library provided by a thirty party. Fortunately, we can still obtain signature hashes from head files which are usually available, and rewrite the library to add these hashes.

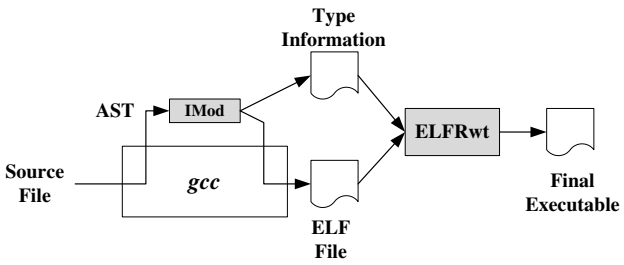


Figure 6. Overview of the implementation

In this case, FPValidator can not validate indirect calls in the dynamic library since the library is not instrumented, but FPValidator can still validate whether other indirect calls, when calling functions in the library, target compatible functions.

B. Evaluation

FPValidator causes extra overhead, including the time overhead of building and executing due to the instrumentation and runtime validation, as well as the space overhead due to the addition of type information. However, we will demonstrate in this subsection that the increased overhead is low.

Adversaries often tend to compromise programs that run for a long time, especially those providing network services, so that they can leave backdoors for future attacks. Therefore we select several server programs as the benchmark, including the Apache http server (*httpd*), the OpenSSH ssh server (*sshd*) and the ProFTPD ftp server (*proftpd*). We measure the overhead brought by FPValidator. The result is shown in Table I.

The column *Functions* and *Calls* show the numbers of function definitions and indirect call statements. The increased compilation and storage cost is in direct proportion to them. The column *Building Time* and *Executable Size* compare the costs of versions built with and without FPValidator. The increased building time is caused by the operations of instrumenting indirect calls. Since the number of indirect calls is small, the increased cost is less than 3%. The increased executable size is mainly cause by the insertion of type information. Because with our method each type can be represented by a single hash value, the space cost is less than 8%. The column *Runtime Overhead* gives the time cost spent on the dynamic function pointer validation. It is determined by how often indirect calls are performed. The result shows it is only about 2%.

C. Security Analysis

FPValidator implements the L3 function pointer validation, which checks the equivalence between the point-to type and target type of function pointers. When using FPValidator, all function entry attacks that violate type equivalence can be detected. To bypass FPValidator, an adversary has

Table I
OVERHEAD

Program	Funcs	Calls	Building Time			Executable Size			Runtime Overhead		
			Normal	FPValidator	%	Normal	FPValidator	%	Total	Validate	%
sshd	995	143	16.445s	16.509s	0.4%	484.9K	502.1K	3.5%	33,000μs	687μs	2.1%
proftpd	1,133	174	10.813s	10.971s	0.9%	734.9K	785.7K	6.9%	82,000μs	1,278μs	1.6%
httpd	2,032	324	78.469s	80.519s	2.6%	809.2K	869.1K	7.4%	115,000μs	2,436μs	2.1%

to find a vulnerability through which he can tamper with a function pointer of the desired type. Thus the possibility of carrying out a successful attack is much lower than that of L1 or L2 validation.

The effectiveness of FPValidator depends on the integrity of the validation code and type information, which are integrated into ELF files. As long as the code segment and the type information are not compromised, the validation mechanism will not be circumvented. Since the code segment and the type information do not change after being loaded, the memory they reside in is marked as read-only during execution. Hence their integrity can easily be verified by lower level software through cryptographic hash functions. For example, the integrity of applications can be verified by the OS kernel, by calculating the hashes of code segment and type information, and comparing them with trusted values. Similarly, the OS kernel itself can be verified by virtual machine monitor. At last, the lowest level software can be verified by some hardware solutions such as the technology proposed by Trusted Computing Group (TCG) [20].

D. The L4 Validation

If the compromised function pointer and the abused function are type equivalent, the attack will not be detected by the L3 validation. In this case the L4 validation is required. The L4 validation requires the precise set of possible targets for each function pointer. Given these sets, the L4 validation can be easily implemented in FPValidator. We just need to replace the type information with the sets, and checks whether the target address is included in the set, instead of type matching.

However, there are some challenges in finding out precise sets. If developers are responsible for giving all valid entry sets, it is a tedious and error-prone task. Furthermore, it is often impossible to list all entries at the development stage, for some functions may be implemented by the third party and loaded at runtime. Some solutions, such as Inlined CFI and WIT, tries to find out precise sets via static analysis [21], [22], but their results turn out to be not precise.

V. RELATED WORK

Manipulating function pointers is an often-used attacking method. A lot of solutions have been proposed to thwart against this kind of attacks. Among them we choose several closely related solutions to compare with.

```
typedef void (*func_t)(void);

int foo(unsigned long parm,
        char * inbuf,
        int len)
{
        func_t fp;
        char buf[16];
        ...
        memcpy(buf, inbuf, len);
        fp = (func_t)parm;
        fp();
        ...
}
```

Figure 7. A buggy function that could be exploited to circumvent PointGuard

CFI is a safety property denoting that a programs execution follows paths of its Control-Flow Graph (CFG) determined in advance. Abadi et al propose an enforcing method, called Inlined CFI [5], based on static binary rewriting. Their method instruments each indirect branch, validating its target according a CFG obtained statically. However, such a CFG can hardly be precise. In order to avoid false-positives, their implementation uses conservative CFGs in which a call instruction may invoke any functions.

WIT [6] enforces CFI through compilation-stage instrumentation. Function pointers and functions, according to the result of static points-to analysis, are labeled with colors. A fragment of validation code is inserted before indirect calls, checking whether the function pointer and the target function have the same color. Since points-to analysis is not precise, it is possible that all functions have the same color in order to avoid false-positives.

Petroni et al propose a virtual machine based method to enforce an approximate of CFI, called state-based CFI (SBCFI) [4]. A monitor, running in the domain 0, validates the global function pointer variables of the OS kernels in other domains periodically. But their method only concerns persistent attacks. Attacks finished within a period could escape from being detected.

Program Shepherding [3] validates indirect branches by dynamic binary translation. The validation code is inserted by a dynamic translator, e.g. Dynamo [23] for their work, into a code block before the block is executed. It checks the targets of indirect branches by censoring a hash table which

Table II
COMPARISON OF RELATED WORK

Solutions	Mechanism	Coverage	Level	Attacks		
				DRA	NFEA	FEA
Inlined CFI	Static Binary Instrumentation	Indirect Branches	L2	√	√	
WIT	Compilation-stage Instrumentation	Indirect Calls	L2+	√	√	√?
SBCFI	VM-based Monitoring	Global Variables	L2	√	√	
Shepherding	Dynamic Binary Instrumentation	Indirect Branches	L2	√	√	
NE Memory	Hardware Support	Indirect Branches	L1	√		
FPValidator	Compilation-stage Instrumentation	Indirect Calls	L3	√	√	√

contains all valid entries.

Non-Executable Memory (NE-Memory), such as Exec Shield [24] and Openwall Linux patch [25], is capable of marking some memory regions (e.g. stack and heap) as non-executable, which usually requiring hardware support [26]. Thus, malicious code that is injected into stack or heap and masquerading as data can not be executed, though an adversary has succeeded in manipulating some function pointers to target it. e-NeXSh uses a software approach, i.e. monitoring all LIBC function and system-call invocations, to create an "effectively" non-executable stack and heap [27].

The comparison of the above work with ours is shown in Table II. Validation mechanism can be roughly divided into instrumentation and monitoring. The former is harder to be bypassed, while the later is more flexible. NE-Memory uses another way to achieve the same effort of instrumentation-based validation. Inlined CFI, Shepherding and NE-Memory work at the binary level and deal with all indirect branches, some of which are derived from, for example, **switch** statements whose targets are actually fixed [28]; while FPValidator and WIT validate all indirect calls whose targets may be changed dynamically. SBCFI only validates global variables, so it fails to detect attacks on local variables and function pointer expressions. The validation levels of these solutions are listed in the column *level*, and the detectable attacks are marked with $\surd$. The level of WIT depends on the precision of static points-to analysis. Currently it could only reach L2 in some cases.

Some solutions try to protect function pointers from being maliciously modified. For example, PointGuard [29] encrypts pointers before writing and decrypts them before reading. The encrypting and decrypting code is also inserted by a compilation-stage instrumentation mechanism. Libsafe [30] and LibsafePlus [31] provide safe versions of exploitable library functions such as *strcpy* and *memcpy*, which could protect function pointers from being modified through buffer overflow attacks. Stack-smashing protector (SSP) [32] achieves this goal by rearranging local variables. However, these solutions can not thoroughly defend against illegal modifications. For example, the protection of PointGuard could be circumvented in some cases. Figure 7 shows a buggy function that could be exploited to circumvent PointGuard. Since PointGuard only encrypts pointers, an adversary could modify *parm* through buffer overflow, and then the modification will be propagated to *fp* at line 9. To provide stronger security, protection and validation of function pointers can be combined to work together.

VI. CONCLUSION

Function pointers are often attacked at the runtime by adversaries to execute malicious code, so it is useful to validate their values on the fly. But the validation performed by existing solutions is not strict enough to detect function entry attacks.

In this paper we propose FPValidator, a new solution capable of dynamically validating the type equivalence of function pointers and target functions, which is able to detect all function entry attacks that manipulate function pointers to invoke incompatible functions. The validation code is inserted by a compilation-stage instrumentation mechanism, bringing no extra burden to developers. We integrate FPValidator into GCC, and the evaluation shows that it is effective and efficient.

Currently we use C as the example language, but FPValidator can also be used in other statically-typed languages. C++, usually viewed as a super set of C, has a more complicated type system. We are going to apply FPValidator to C++ in our future work.

REFERENCES

[1] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20 – 27, 2004.

[2] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *the 7th Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 2006, pp. 147 – 160.

[3] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," in *11th USENIX Security Symposium*. San Francisco, California: USENIX Association, 2002, pp. 191 – 206.

[4] J. Nick L. Petroni and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," in *the 14th ACM Conference on Computer and Communications Security (CCS'07)*. Alexandria, Virginia, USA: ACM, 2007, pp. 103 – 115.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *the 12th ACM Conference on Computer and Communications Security (CCS'05)*, Alexandria, VA, USA, 2005, pp. 340 – 353.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 263–277.

[7] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, 1994.

[8] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2001, pp. 54–61.

[9] C. Chambers, D. Ungar, and E. Lee, "An Efficient Implementation of Self, A Dynamically-Typed Object-Oriented Language Based On Prototypes," *ACM SIGPLAN Notices*, vol. 24, no. 10, pp. 49–70, 1989.

[10] M. Chang, M. Bebenita, A. Yermolovich, and A. Gal, "Efficient Just-In-Time Execution of Dynamically Typed Languages Via Code Specialization Using Precise Runtime Type Inference," Donald Bren School of Information and Computer Science, University of California, Irvine, Tech. Rep., 2007.

[11] C. Flanagan, "Hybrid Type Checking," in *the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, USA, 2006, pp. 245 – 256.

[12] S. Fagorzi and E. Zucca, "A Calculus of Components with Dynamic Type-Checking," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 182, pp. 73 – 90, 2007.

[13] R. B. Findler and M. Felleisen, "Contracts for Higher-order Functions," *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 48 – 59, 2002.

[14] U. Erlingsson, "Low-Level Software Security: Attacks and Defenses," in *Foundations of Security Analysis and Design IV, LNCS 4677/2007*. Springer Berlin, 2007, pp. 92–134.

[15] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools 2nd*. Addison-Wesley, 2006.

[16] A. M. Zaremski and J. M. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 146 – 170, 1995.

[17] M. V. Aponte and R. D. Cosmo, "Type Isomorphisms for Module Signatures," in *the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*. Springer-Verlag, 1996, pp. 334 – 346.

[18] S. Jha, J. Palsberg, and T. Zhao, "Efficient Type Matching," in *the 5th International Conference on Foundations of Software Science and Computation Structures*. Springer-Verlag, 2002, pp. 187–204.

[19] Wikibooks, "GNU C Compiler Internals," 2008. [Online]. Available: http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals

[20] TCG, "TCG Specification Architecture Overview," August 2007.

[21] D. Liang and M. J. Harrold, "Efficient Points-to Analysis for Whole-program Analysis," in *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. London, UK: Springer-Verlag, 1999, pp. 199–215.

[22] N. Heintze and O. Tardieu, "Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C code in a Second," *SIGPLAN Notices*, vol. 36, no. 5, pp. 254–263, 2001.

[23] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," in *the ACM SIGPLAN 2000 Conference On Programming Language Design And Implementation*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 1 – 12.

[24] "Exec Shield. http://people.redhat.com/mingo/exec-shield."

[25] "Linux kernel patch from the Openwall Project. http://www.openwall.com/linux/."

[26] "Intel 64 and IA-32 Architectures Software Developer's Manual," November 2007.

[27] G. S. Kc and A. D. Keromytis, "e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," in *the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005, pp. 286 – 302.

[28] C. Cifuentes and M. V. Emmerik, "Recovery of Jump Table Case Statements from Binary Code," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 171 – 188, 2001.

[29] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," in *the 12th USENIX Security Symposium*, 2003, pp. 91 – 104.

[30] A. Baratloo, T. Tsai, and N. Singh, "Libsafe: Protecting Critical Elements of Stacks," December 25 1999.

[31] K. Avijit, P. Gupta, and D. Gupta, "TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection," in *the 13th USENIX Security Symposium*, 2004.

[32] H. Etoh, "GCC Extension for Protecting Applications from Stack-smashing Attacks (ProPolice)," 2003. [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/