Mapping to Bits: Efficiently Detecting Type Confusion Errors

Chengbin Pang
National Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology
Nanjing University, China
binpangsec@smail.nju.edu.cn

Yunlan Du
National Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology
Nanjing University, China
mf1733007@smail.nju.edu.cn

Bing Mao
National Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology
Nanjing University, China
maobing@nju.edu.cn

Shanqing Guo
Key Laboratory of Cryptologic Technology and
Information Security, Ministry of Education
Shandong University, China
guoshanqing@sdu.edu.cn

ABSTRACT

The features of modularity and inheritance in C++ facilitate the developers' usage, but also give rise to the problem of type confusion. As an ancestor class may have a different data layout from its descendant class, a dangerous downcasting operation from the ancestor to its descendant can lead to a critical attack, such as control flow hijacking, out-of-bounds access to neighbor memory area, etc. As reported in CVE, such vulnerabilities have been found in various common-used software, including Google Chrome, Firefox and Adobe Flash Player, and have a trend of increase in recent years. The urgency of addressing type confusion problems quickens the pace of researchers coming to corresponding solutions. However, the existing works either handle the problem partially, or suffer from the high performance and memory overhead, especially to the large-scale projects.

We present Bitype to check the validity explicitly when a type is downcasting to another, maintaining high coverage and reducing overhead and compilation time massively. The core of our design is a Safe Encoding Scheme, which encodes all of the classes by mapping them to bits. With this scheme, Bitype treats the classes and their safe convertible classes as codes and verifies typecastings in an `xor` operation, both decreasing the performance overhead of check and the memory overhead. Besides, we implement a Clang Tool to avoid the repeated collection of inheritance relationships and deploy a two-level lookup table to trace objects efficiently. Evaluated on SPEC CPU2006 benchmarks and Firefox browser, Bitype shows a slightly higher coverage of typecasting compared to the state-of-the-art HexType[22], but reduces the performance overhead by 2 to 16 times, the memory overhead by 2 to 3 times, the compilation time by 21 to 223 times. As a result, our solution is a practical and efficient typecasting checker for commodity software.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Systems security*;

KEYWORDS

Type confusion, Typecasting, Safe Encoding Scheme, Downcasting

ACM Reference Format:
Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. 2018. Mapping to Bits: Efficiently Detecting Type Confusion Errors. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3274694.3274719

1 INTRODUCTION

C++ is widely used in software development, due to its design highlights of the modularity of object-oriented programming, the efficiency of low level memory access and performance. Some well-known applications, such as Google Chrome, Firefox, and the Oracle Java Virtual Machine, are all successful programs in C++ language.

However, C++ suffers from memory and type safety, which leave programs with huge risks. Among them, type confusion is one of the most common and dangerous vulnerabilities in C++, as it mixes the above two safety problems.

Type confusion occurs when the program intercepts an object of a type as an object of another different type in a process of unsafe typecasting. It is common that a program casts an object of a parent class to a descendant class. Thus, there exists a possibility of unsafe typecasting because the parent class may lack some fields or virtual functions of the descendant class. If the program subsequently uses such fields or functions, it may use data, say, as a regular field in one context or as a virtual function table(vtable) pointer in another. When attackers abuse a type confusion vulnerability, they can easily perform arbitrary memory read/write, sometimes even gain a higher privilege from it.

The statistics of CVE show a wide spread of type confusion vulnerabilities in popular applications, including Google Chrome (CVE-2017-5116), Adobe Flash Player (CVE-2018-4944), Foxit PDF Reader (CVE-2018-9942). Those exploitable type confusion bugs exactly do harm to software safety, and need to be addressed right away.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '18, December 3–7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274719

Checking typecastings at runtime is the current method to solve type confusion problems. If we choose to identify objects through existing fields embedded in the objects such as vtable pointers[18, 29, 38, 43], we will definitely lose control of non-polymorphic objects which have no vtable pointers, and have to blacklist the unsupported classes manually in case of an unexpected crashes. Therefore, another way of leveraging disjoint metadata[19, 22, 25] to identify objects seems better, and is widely-used in recent researches.

It is obvious that the current solution has two main sources of overhead: (1) maintaining metadata when allocating and deallocating objects, (2) explicit type checks at cast locations. Most researchers[19, 22, 25] focus their optimization on the first point. CaVer[25], which uses red-black trees to trace global and stack objects, is too slow and expensive to adapt to present production systems. HexType[22] tries to employ a hash table to reduce object tracing overhead, but it brings a hash collision problem especially in large projects where more objects need to be traced. TypeSan[19] designs an efficient metadata storage service based on a shadowing scheme to look up types from pointers, yet only supports tcmalloc[33] (the allocator used by the Chrome browser and other Google products). Although all the state-of-the-art detectors[19, 22, 25] endeavor to reduce the overhead from maintaining metadata, the performance and the memory overhead and the compilation time are still too high. In particular, the larger the program, the greater the cost.

We propose **Bitype** as an always-on type checker with high coverage and massively reduced performance and memory overhead and compilation time. Bitype is aimed at making explicit type checks for software and programs to avoid the risks from type confusion, and especially overcomes the prohibitive costs of large programs.

Bitype puts its effort to the reduction of the costs mainly from three aspects: (1) It designs a Clang Tool[37] to collect the inheritance relationships in a global scope. When a module is loaded, Bitype no longer needs to compute the inheritance relationships repeatedly, so that the compilation time will be declined significantly. (2) Metadata is stored in a two-level lookup table, which requires only 3 memory reads to look up a type. (3) Compared with previous work, Bitype realizes extra optimization to the process of verification of types. A novel Safe Encoding Scheme is introduced to label whether a class can convert to another class safely. With this encoding scheme, Bitype verifies the convertibility only by an easy `xor` operation in a time complexity of $O(1)$. Thus, Bitype reduces the costs of large programs to an acceptable level while ensuring high coverage.

We have implemented a prototype of Bitype for Linux on top of LLVM 7.0[3]. We have evaluated Bitype with the SPEC CPU2006 C++ programs and the Firefox browser. Compared with HexType, Bitype shows a 2 - 16 times reduction in performance overhead for SPEC CPU2006 benchmarks, and 2 - 6 times reduction for Firefox browser, 2-3 times reduction in memory overhead, and 21-223 times reduction in compilation time overhead.

Our major contributions list as follows:

- We design an explicit and efficient type checker Bitype to examine the typecasting whether or not safe at runtime, and thus protect type and system safety of product software from type confusion bugs.
- We implement a useful Clang Tool for inheritance information collection in global scope, which avoids long compilation time from repeated collection work.
- We design an original class Safe Encoding Scheme to accelerate the type verification. It is a new direction to actually reduce the overhead of the verification process.
- We evaluate our type checker on SPEC CPU2006 C++ benchmark and current commodity software, Firefox browser and the test results show that our tool has high coverage, low performance and memory overhead and low complication time.

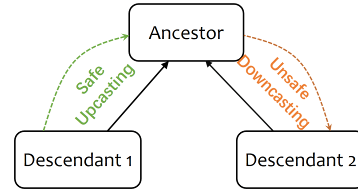


Figure 1: Visualization of an example C++ type hierarchy, showing the directions of (safe) upcasting and (unsafe) downcasting.

2 BACKGROUND

In this section, we first introduce the inheritance and cast operations in C++ as basic knowledge. We then give a detailed explanation of type confusion. At last, we discuss about previous defenses against type confusion.

2.1 C++ Inheritance and Cast Operations

C++ is a general-purpose programming language, which has imperative, object-oriented and generic programming features, and provides facilities for low-level memory manipulation.

Inheritance in C++ allows one data type to acquire properties of other data types. If base classes are declared as virtual, it is virtual inheritance which ensures that only one instance of a base class exists in the inheritance graph, avoiding some of the ambiguity problems of multiple inheritance. Multiple inheritance is a C++ feature not found in most other languages, allowing a class to be derived from more than one base class; this allows for more elaborate inheritance relationships.

C++ also allows a pointer of one type to cast into a pointer of another one, which means the pointed object will be treated as another type. This kind of conversion happens frequently between a descendant class and an ancestor class in programming, because it makes the code reuse more easily. As the descendant class is derived from its ancestor classes, the data layout is such that an object from the derived class containing the fields of its parent classes at the same relative offsets. Thus, in certain cases, the cast between a descendant class and an ancestor class can be valid. That is, we should confirm if a typecasting is safe or not.

We define a type conversion from a descendant class to its ancestor class as an *upcasting*. It is always safe and permissible, since

the members in the descendant class are a superset of the members in the ancestor class and the data fields are enough for layout. Relatively, we define the typecasting from the ancestor class to its descendant class as an *downcasting*. However, when the parent lacks any member of its child, the downcasting is not safe, and even does harm to the underlying system. Figure 1 shows upcasting and downcasting between an ancestor and its descendants.

As a result, C++ provides some cast operations to help programmers to check the safety of downcastings, including three types: `static_cast`, `dynamic_cast`, and `reinterpret_cast`. A `static_cast` checks the type of object only at compile time, and is constrained by the static nature leading to the ignorance of runtime check. If the underlying type observed at run time is different from what it expected in the source code, static checks can do nothing and type confusion problems may exist. A `dynamic_cast`, on the other side, fills the gap in static check while preforming the runtime check, but pays a great price of overhead as an exchange. The expensive cost comes from Run Time Type Information (RTTI), which encodes all type related information. A `reinterpret_cast` is used to convert between any two types by force regardless of compatibility. Since the fundamental cast operations cannot meet the demand of eliminating type confusion, we should do some extra improvement to make software safer.

2.2 Type Confusion

Here we give an example to explain the process of type confusion in detail.

Figure 2(a) is a code example of type confusion. The Child class is derived from the Parent class (line 4), and a `static_cast` operation allows the compiler to treat the object of Parent `*P` as a type of Child (line 14). This downcasting from Parent to Child is definitely unsafe, because the access scope of Child, which contains an extra virtual pointer and an integer, is larger than the one of Parent as Figure 2(b) shows. Thus, when the attacker modifies the integer `y` of the downcast Child (line 16), it will result in a memory safety violation. In a more serious situation, if the program makes use of the attacker-controlled vptr in the scope of Child (line 18), its control flow will be hijacked.

Type confusion is getting more and more attention from developers and researchers. Figure 3 counts the number of type confusion vulnerabilities reported on CVE[1] from 2010 to 2018, and depicts a trend of growth. In addition to its quantitative increase, type confusion also has a more serious impact on more popular applications, as is shown in Google Chrome (CVE-2017-5116), Adobe Flash Player (CVE-2018-4944), Foxit PDF Reader (CVE-2018-9942).

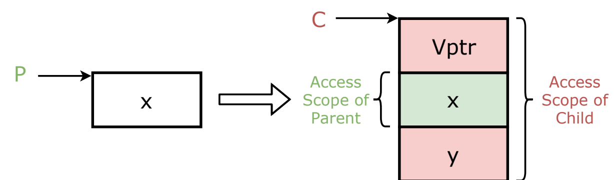
2.3 Defenses against Type Confusion

To solve the matter of type confusion, several mechanisms have been put forward to detect the unsafe typecastings and protect programs.

Earlier, researchers tried to identifiy objects through embedded vtable pointers, such as UBSan[29] which instruments static casts to turn them into dynamic casts. However, as we mentioned before, this measure does not support non-polymorphic classes which need manual blacklisting, and dynamic check itself is too slow to suit a large amount of usage.

```
1  class Parent{
2     int x;
3  };
4  class Child : public Parent{
5  public:
6     int y;
7     virtual void print(){
8        cout << "I am in Child Class" << endl;
9     }
10 };
11 int main(int argc, char**argv){
12    Parent *P = new Parent();
13    // Type confusion
14    Child *C = static_cast<Child*>(P);
15    // Memory safety violation
16    C->y = 0x1234;
17    // Control-flow hijacking
18    C->print();
19 }
```

(a)



(b)

Figure 2: An example of type confusion. (a) is the code of the example, (b) is the memory scope of *P and *C.

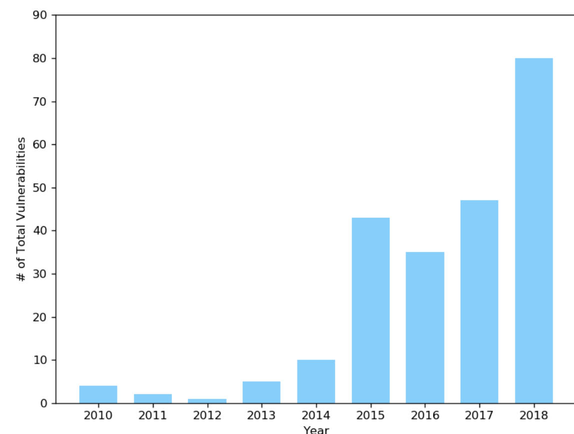


Figure 3: The number of type confusion vulnerabilities reported on CVE in recent years.

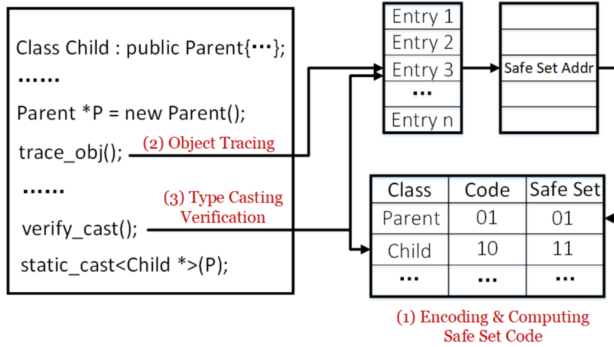


Figure 4: The overview of Bitype. Bitype consists of several modules that encode and compute safe set code and insert object tracing and typecasting instrumentation to verify typecasting operation.

CaVer[25] improves the identification by leveraging disjoint metadata to support non-polymorphic classes, and is the first typecasting detection tool. Unfortunately, the overhead is still prohibitively high and the checking coverage is low.

TypeSan[19] designs an efficient metadata storage service based on a shadowing scheme to look up types from pointers. Although it reduces the performance overhead and increases detection coverage compared to Caver, the overall coverage is still low, and only supports tcmalloc allocator.

HexType[22] tries to employ a hash table to reduce object tracing overhead, but it brings a hash collision problem especially in large projects where more objects need to be traced. In addition, its long compilation time is a defect to large programs.

From the four aspects of coverage, performance overhead, memory overhead and compilation time, we introduce a new type checker Bitype to improve the current type confusion detection work.

3 THREAT MODEL

We assume that the program contains type confusion vulnerabilities which may be exploited by attackers. We further assume that the attacker may read arbitrary memory but unable to perform arbitrary memory writes. Our design is aimed at fast and explicit detecting unsafe type confusion and does not rely on the hiding information from the attacker. We also suppose other vulnerabilities which do not arise from type confusion, such as integer overflow, and memory corruption, can be defensed by other relative protection tool.

4 BITYPE DESIGN AND IMPLEMENTATION

Bitype is an efficient and explicit type confusion detector based on Clang/LLVM compilers in C++ programs. Given a source code as input, the Bitype-optimized compiler generates a Bitype-hardened binary as a result. Once a type confusion error is detected, the program is terminated by force and sends a bug report.

4.1 Overview

Figure 4 illustrates the overview of Bitype. When a source code is input to the compiler, a Clang Tool first analyzes abstract syntax tree to obtain the inheritance relationships among all the classes and indexes these classes. Then, we encode all the classes according to the encoding scheme, and compute the safe set codes which represent the sets of safe types that can be validly converted to according to the inheritance relationships. The encoding and computing rules will be introduced later in section 4.2.

After the preprocess of collecting type relationship information, we need to obtain the true types of each allocated object during runtime. A `trace_obj()` method is instrumented after each statement of allocating a new object. In this method, we establish and maintain a two-level lookup table to map the runtime address of the object to its safe set code. Section 4.3 will explain in further how the lookup table plays a part in object tracing.

Bitype verifies the typecastings within our unique encoding scheme. When a `static_cast` tries to convert an object from a source type into a target type, a `verify_cast()` method is called to compare the code of the target type with the safe set code of the source type. It is an O(1) time complexity verification process and reduces the performance overhead to a certain extent. In section 4.4, the process of verification will be presented.

4.2 Safe Encoding Scheme

In order to verify whether the conversion of the type is safe or not, we elaborate a Safe Encoding Scheme to deal with the problem in a more efficient way.

For preparation, we choose to use a Clang Tool to collect those inheritance relationships, instead of LLVM. In TypeSan and HexType, LLVM is used to collect inheritance relationships, which may omit some type information during the transition from source code to LLVM IR. Another two factors also contribute to the preparation with Clang Tool. On the one hand, a preprocess of collection can avoid the following repeated computation and as a result massively reduce compilation time. On the other hand, since our Safe Encoding Scheme needs indexing all the classes globally, we need to get all the classes in the whole project. Figure 5(b) shows an example of relationships of the code in Figure 5(a).

4.2.1 Encoding Classes. In our Safe Encoding Scheme, we label each class found by the Clang Tool with an index number from 1 to n, as shown in Table 1. We integrate the step of indexing classes in the Clang Tool as well. Traversing the abstract syntax tree, the Clang Tool searches all the classes and inheritance relationships. Specially, it obtains the definitions of the classes and gets all their base classes. Then it divides classes into several families by the inheritance information and compound relationship, and indexes all the classes according to the rules mentioned later.

We define $Index_{Class}$ (1<= $Index$ <= n, n is the number of the classes in this family) to describe the index of a certain class $Class$, and $Code_{Class}$ as its unique code.

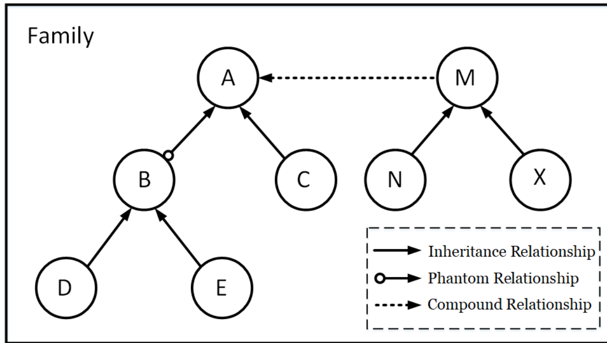
$$Code_{Class} = 1 << (Index_{Class} - 1)$$

Taking compatibility into consideration, we should concern phantom classes in addition. If the data layout of a derived class

```
//class definition
class A{...};
class B : public A{ void func{...};};
class C : public A{...};
class D : public B{...};
class E : public B{...};
class M {A a; ...};
class N : public M{...};
class X : public M{...};

//cast example
void casting(){
   M *m = new M();
   //downcast, unsafe
   static_cast<B*>(&(m->a));
}
```

(a)



(b)

Figure 5: (a) shows an example code with several inheritance relationships. We assume that, except for phantom class B, A cannot safely downcast to any other classes in this family. (b) is the relationships of the code in (a).

is the same as its base class, the derived class is a phantom class. As such classes are used frequently when programming, they are treated as safe typecastings by our mechanism. In Safe Encoding Scheme, a phantom class is labeled the same index as its base class to downcast validly. Class B in Figure 5, for example, is a phantom class derived from class A, because it only contains a function which does not make the data layout of class B different from the one of A. Thus, Table 1, a presentation of encoding results, shows that class A and B share the same class index 1.

Figure 5(a) also describes a compound relationship between class A and M. Since the first member variable of the object `*m` of class M is an object of class A (line 7), the address of `m->a` is exactly where `m` points to. In order to build a connection between such compound classes like A and M, the two classes themselves and all their descendants and ancestors are regard as a family. When `m->a` downcasts to a type of B, we can check all the ancestors of A and M to protect this typecasting.

Class Name	Class Index	Code	Safe Set Code
A	1	0000 0001	0000 0001
B	1	0000 0001	0000 0001
C	2	0000 0010	0000 0011
D	3	0000 0100	0000 0101
E	4	0000 1000	0000 1001
M	5	0001 0000	0001 0000
N	6	0010 0000	0011 0000
X	7	0100 0000	0101 0000

Table 1: An example of how to encode classes in Figure 5 and compute their safe set codes according to the Safe Encoding Scheme.

Besides, due to the tremendous number of classes in large-scale programs, the division by families also helps to shorten the coding length to relief the problem brought by the too large indexes. Each family encodes from 1, and deals with the normal and compound relationships, and the phantom classes. Although there exist repeated indexes caused by the family division in an overall view, they will not disturb each other because `static_cast` prohibits the typecastings across the family when compiling.

Algorithm 1 Algorithm of Computing Safe Set Code

Input:
The calculated class, *Class*;
The set of the calculated class's ancestors, $SafeSet_{Class}$;
Output:
The safe set code of calculated class, SSC_{Class};
1: Initialize $SSC_{Class} = 0$
2: $Code_{Class} = 1 << (Index_{Class} - 1)$
3: $SSC_{Class} = SSC_{Class} \lor Code_{Class}$
4: **for** each $C \in SafeSet_{Class}$ **do**
5: $Code_C = 1 << (Index_C - 1)$
6: $SSC_{Class} = SSC_{Class} \lor Code_C$
7: **end for**
8: **return** SSC_{Class}

4.2.2 Computing Safe Set Code. The Safe Set of a class is defined as a set of its ancestor classes that can be safely converted to. The Safe Set Code is then computed to represent all the classes in Safe Set. SSC_{Class} is used to describe the Safe Set of *Class* in the form of code. Algorithm 1 illustrates the process of computing SSC_{Class}.

What we need to input are the class to be calculated, and its Safe Set collected by Clang tool during the preprocess. After encoding *Class* itself and the classes in its Safe Set, we `or` all the above codes iteratively to compute the SSC_{Class}.

Take the class D in Figure 5 as an example. Class D is derived from class B, while B is derived from A. Thus, the Safe Set of D consists of A and B, both encoded as 001 (for short) in Table 1 as B is a phantom class. Moreover, D, the code of which is 100(for short), can surely cast to D itself, so we compute SSC_D by $100 \lor 001 = 101$.

Section 4.3 introduces an additional computation rule aimed at dealing with the more complex situation with compound relationship.

When we obtain a Safe Set Code, we can determine the safety of a typecasting simply through calculating $SSC_{SourceClass} \oplus Code_{TargetClass}$. As the result is not equal to $SSC_{SourceClass}$, when the result is smaller, this typecasting will be considered safe. Otherwise, if the result is larger, it will be considered as a type confusion error and reported to us. More detailed and complete explanations about verification computing rules is presented in section 4.4.

In the implementation, *SSC* is one-byte-aligned, and is of the same size in the one family. If a family has the maximum class index of n, the size of *SSC* in this family will be $\lfloor (n-1)/8+1 \rfloor$ bytes. We store the Safe Set Code in a global region which is read-only.

4.3 Object Tracing

During the process of object tracing, we need to record the true type information of the allocated object at runtime. RTTI, which is used to retrieve all type information in `dynamic_cast`, has prohibitive runtime overhead and no support to non-polymorphic objects, and even blows up the sizes of compiled binary.

Thus, we give up utilizing RTTI, and attempt to build the relationship between the object address and its type information. We assume that *addr* is the start address of an instance of *Class*. An Address Safe Set Code $ASSC_{addr}$ is defined to represent the Safe Set Code of the object in *addr*. When the object of *Class* is allocated, we can get its start address and type information. We then have

$$ASSC_{addr} = SSC_{Class}$$

However, the compound relationships, such as the example in Figure 5, make the tracing more complex. After we allocate an object of class M in line 13, what in the start address of $^*m$ can be an instance of class A or M. Thus, when we decide the Safe Set of the object in address m during the process of object tracing, both Safe Set of A and M should be concerned. The object in m can be safely converted to all the classes in the Safe Set of A and the Safe Set of M. We can compute $ASSC_m$ as follow:

$$ASSC_m = SSC_M \vee SSC_A$$

Furthermore, if several compound classes all have zero offset to the data layout of *Class*, named *CompClass*1, *CompClass*2, ..., *CompClassN*, the computation rule of $ASSC_{addr}$ need to be revised to

$$ASSC_{addr} = SSC_{Class} \vee SSC_{CompClass1} \vee SSC_{CompClass2} ... \vee SSC_{CompClassN}$$

On the one hand, the compiler allows the further check of cast operations only when the target class has an inheritance relation with the source class. On the other hand, our encoding rules ensure the classes with a compound relationship are divided into the same family and share the same set of codes. Thus, the $ASSC_{addr}$ is surely computable.

We instrument a `trace_obj()` method after allocating a new object and store their relationship with a two-level lookup table. Previous methods to track object have several limitations. TypeSan stores the metadata table in shadow memory in a fixed address, so

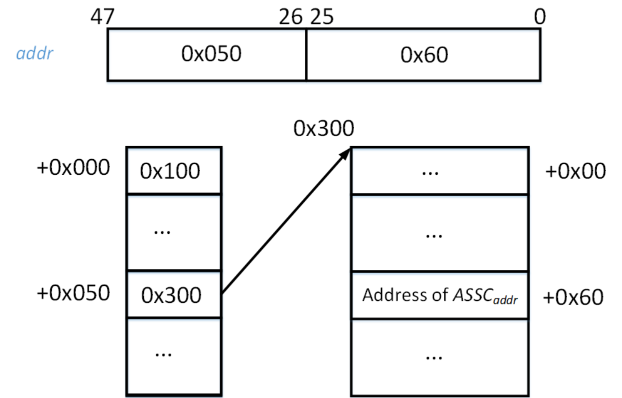


Figure 6: An example of the two-level lookup table that connect the address of an object and its *ASSC*. The entry for the top level table points to the second level table and the entry for the second level table is the address of *ASSC*.

the compatibility of program will be influenced if the application reuses such areas. Also, TypeSan does not delete the related information when an object is freed. While HexType implements a hash table for storage, a smaller table will possibly lead to hash collision and a bigger one will bring expensive memory overhead.

Previous work CFIXX[6] has proved that two-level lookup table is suitable for storage of C++ object metadata. Our two-level lookup table overcomes the above shortcomings, shown as Figure 6. In Linux x86_64, only the lower 48 bits are used for virtual address space. Splitting the address of the object into two parts, we take the high order 22 bits as the index of the first level table, while the other 26 bits as the index of the second one.

In Figure 7, *addr* is the object address we get during allocation, with *0x050* in the high order 22 bits and *0x60* in the low order 26 bits. The first level table stores the entry *0x300* at an offset of *0x050*. The entry *0x300* is then extracted as the entry address of the second level table. Next, we can find the address of $ASSC_{addr}$ information which is stored in the second level table with an offset of *0x60*. Up to now, the relationship between the object address addr and its Address Safe Set Code in $ASSC_{addr}$ has been built. This method takes only 3 memory reads to fetch information in $O(1)$ time complexity.

4.4 Typecasting Verification

The last step of detecting a potential type confusion error is to verify whether the typecasting is safe at cast location. The state-of-the-art checkers all traverse the data structures stored information of Safe Sets, with a time complexity of $O(logN)$ or even $O(N)$, N is the number of classes in the Safe Set.

Our Safe Encoding Scheme significantly speeds up the verification of typecastings. In Figure 7, as we have computed $ASSC_{pd}$ to represent the Safe Set of the object in address `pd` at the time of allocation, the safe upcasting from `*pd` to `*pa` means the address `pd` is equal to `pa`, and $ASSC_{pa}$ is $ASSC_{pd}$ in nature. Unlike upcasting, a probably dangerous downcasting need to be checked by the instrumented `verify_cast()` method. When the object in `pa`

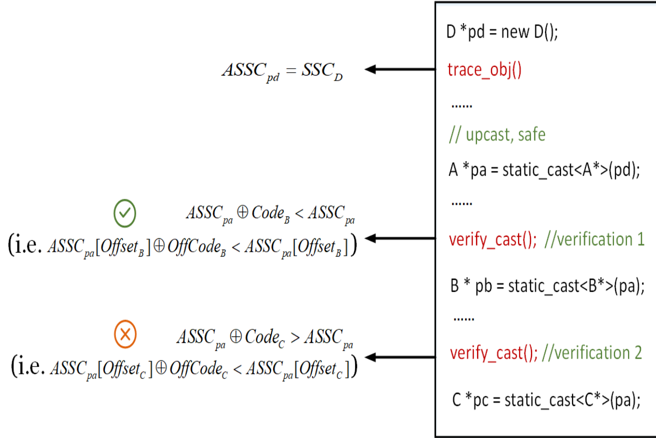


Figure 7: An example of typecasting verification. The class declarations are in the figure 5(a) and the safe set codes of the class are in the table 1.

is going to static cast to a type of B, we fetch the corresponding $ASSC_{pa}$ from pa via the two-level lookup table mentioned before, and $Code_B$ from $Index_B$ in Table 1.

$$ASSC_{pa} = SSC_D = 101$$
$$Code_B = 001$$
$$ASSC_{pa} \oplus Code_B = 100 < 101$$

According to the verification rules of Safe Encoding Scheme, this downcasting from class A to B is considered as a safe operation. In fact, B is the phantom class of A, thereby the verification result is proved correct. We can similarly compute the example of Verification 2 in Figure 7.

$$Code_C = 010$$
$$ASSC_{pa} \oplus Code_C = 111 > 101$$

As the result of xor is larger than $ASSC_{pa}$, the downcasting from class A to C is illegal.

Instead of operating the full-length $ASSC$ and $Code$, we simplify the operation to an xor of one byte. We assume that tc is the target class in the downcasting to be verified, and the source object is stored in address $addr$ with the code $ASSC_{addr}$. Due to the rule of encoding classes, we only need to verify the $No.Index_{tc}$ bit of $ASSC_{addr}$ is *1* or not. We define $Offset_{tc}$ to discribe the *byte* offset of $Index_{tc}$, $OffCode_{tc}$ to extract the byte which contains *1* in $Code_{tc}$ by $Offset_{tc}$, calculated as follow.

$$Offset_{tc} = \lfloor (Index_{tc} - 1)/8 \rfloor$$
$$OffCode_{tc} = 1 << ((Index_{tc} - 1)\%8)$$

We can calculate the $Offset_{tc}$ and $OffCode_{tc}$ during the process of compilation, thus the runtime overhead can be reduced. Then we verify the typecasting with $ASSC_{addr}[Offset_{tc}]$, which represent the $No.Offset_{tc}$ byte of $ASSC_{addr}$, instead of a complete $ASSC_{addr}$.

$$ASSC_{addr}[Offset_{tc}] \oplus OffCode_{tc} > ASSC_{addr}[Offset_{tc}]$$

is unsafe.

$$ASSC_{addr}[Offset_{tc}] \oplus OffCode_{tc} < ASSC_{addr}[Offset_{tc}]$$

is safe.

Our Bitype massively reduces the overhead from explicit type checks at cast locations to $O(1)$ time complexity. In the process of verification, it performs both efficiency and correctness.

4.5 Optimization

In order to achieve lower performance and memory overhead, we implement previous[19, 22, 25] works' optimization(i.e. only verifying unsafe casting, only tracing unsafe object, etc.). Besides, we optimize Bitype by only indexing the target classes, tracing objects of cast-related classes and increasing the granularity of tracing from one byte to eight bytes.

4.5.1 Only Indexing the Target Classes. If a class is not any target classes of all downcastings, it doesn't matter whether it is in the Safe Set or not, as the Safe Set contains the safe classes that can be cast to. Thus, we zero set the indexes and Safe Set Codes of those classes, and only pay attention to the target classes when encoding. As a result, we may decrease the maximum index in a family, and reduce the binary size efficiently.

4.5.2 Only Tracing Objects of Cast-Related Classes. It is a basic idea of optimization to only trace unsafe objects, i.e. objects subject to typecasting. Previous works, such as HexType, apply this measure based on LLVM, focusing on tracking cast-related classes, which have inheritance relationships with target classes. However, if an object of class M consists an instance of another class A with no inheritance relation, HexType will treat the unrelated A as a cast-related class when M is a target class. Thus, when HexType handles some large programs, a lot of unrelated classes will be mistaken as cast-related classes.

	HexType	Bitype	dec%
dealII	126.4	124.4	1.6
xalancbmk	327.6	89.1	72.8
omnetpp	223.5	217.3	2.8
soplex	13.2	4.5	65.9
average			35.8
ff-octane	511.0	195.7	61.7
ff-drom-js	1301.7	876.5	32.7
ff-drom-dom	2267.3	1290.1	43.1
average			45.8

Table 2: The number(million) of traced objects for HexType and Bitype.

Since we utilize the Clang tool to collect all inheritance relationships and all target types in the downcastings, we tend to avoid tracing mistaken cast-related classes. Table 2 lists the programs which have cast-related classes, and shows a reduction in the number of Bitype-traced objects compared with the Hextype-traced objects. The average rate of decrease is 35.8% for tests on the SPEC

CPU2006 C++ benchmarks, and is 45.8% for Firefox. The performance overhead has a corresponding reduction due to the less objects to trace.

4.5.3 Increasing the Granularity of Tracing from One Byte to Eight Bytes. When we trace an object, we build a relationship between a one-byte content at object address *addr* to the 8-byte address of $ASSC_{addr}$. This mapping from one byte to eight bytes costs too much memory space and need to be improved.

We count the number of cast-related classes whose sizes are at least 8-byte aligned in their data layout. 99.54% in Firefox and 100% in all programs in SPEC CPU2006 C++ which have cast-related classes, are 8-byte aligned or even 16-byte aligned. This fact, however, inspires us to make an eight-to-eight mapping during tracing the objects to save nearly seven times the memory space. That is, we trace every eight bytes instead of one byte, and the eight-byte content starting from the address *addr* is connected to the 8-byte address of $ASSC_{addr}$.

Here comes another problem. If two cast-related types of one-byte objects are stored in the same eight bytes, ASSC will be covered by the last traced object. Nevertheless, the fact of nearly 100% at least 8-byte-aligned classes indicates that the probability of such condition is rather low. In order to address this problem, we still have to ensure that the start address of object of each cast-related class is 8-byte aligned. We consider the solution from three aspects: the heap, the stack and the global objects.

When allocating objects on the heap, most allocators equipped on 64-bit machines, such as ptmalloc[16], tcmalloc[33] and jemalloc[12], allocate a size of at least 8-byte-aligned. Thus, it is assured that the allocation on the heap is 8-byte aligned.

As to stack and global objects, we turn cast-related classes which are less-than-8-byte aligned to 8-byte aligned by force with LLVM. Due to the low probability, its impact on memory overhead is limited. While we only force the allocated space to be 8-byte aligned during allocation, it is ABI compatible.

Since the objects of cast-related classes are at 8-byte aligned start address, we can then check whether the member variables in every eight bytes are of different cast-related classes during compilation. If there exist two or more cast-related classes in the eight bytes, we will compute their Safe Set Codes to ASSC of the start address.

4.6 Implementation

We have implemented Bitype based on LLVM Compiler infrastructure project[3] (version 7.0). Bitype contains several parts of Clang[36], Clang Tool[37], LLVM Passes[31] and compiler-rt runtime library[30], including 4829 lines of code in total.

As is shown in Figure 8, when the source code is input, the Clang Tool firstly collects the inheritance relationship and cast-related classes, and indexes the target classes in downcastings. Clang is modified to instrument typecasting verification before every downcasting. We also complete LLVM passes to carry out our Safe Encoding Scheme and instrument object tracing. Bitype finally generates a Bitype-hardened binary. At runtime, the instrumentation successively invokes the Bitype's runtime library functions to trace objects and verify typecastings.

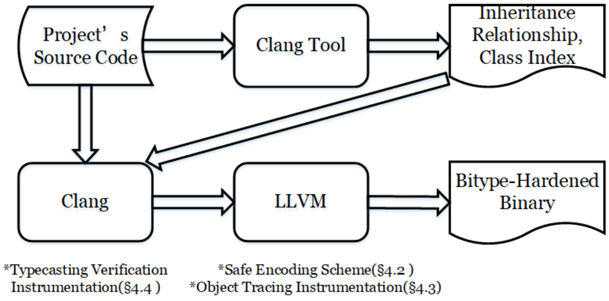


Figure 8: The overview of Bitype implementation. Bitype consists of a Clang Tool which collects class informations, several compiler passes in Clang and LLVM which insert object tracing and typecasting instrumentation, and a corresponding runtime library.

5 EVALUATION

To evaluate the coverage, performance overhead, memory overhead and compilation time of Bitype, we implement it on 7 C++ benchmarks from SPEC CPU2006[8] which are mentioned in TypeSan and HexType, and Octane[17] and Dromaeo[13] benchmarks of Firefox browser(version 53.0.3)[14]. All the evaluation is performed in the Ubuntu 16.04 LTS with a quad-core 4.2GHz CPU(Intel i7 7700K), 1TB HDD and 16GB RAM.

We also attempt to test both HexType and TypeSan on SPEC CPU2006 and Firefox with the same configuration at the aim of comparison. We run the HexType-compiled Firefox program by ourselves, while citing the results of TypeSan in [19] due to it compiling a lower version of Firefox.

5.1 Coverage

As the total amount of downcasting operations in a program is fixed, we can count the number of verified typecastings and compute the coverage to evaluate the effectiveness of Bitype. Similar to HexType, Bitype also handles `placement new` and `reinterpret_cast` to increase coverage.

In SPEC CPU2006 benchmarks, the coverages reach 100% in omnetpp, dealII and soplex, and 99.8% for xalancbmk. In Firefox benchmarks, the average coverage 83.1% of Bitype is slightly higher than HexType's 79.9%, and much higher than TypeSan's 27.8%. The slightly higher coverage compared to HexType is resulted in that when the HexType handle the `reinterpret_cast` and `placement new` operations, they only trace the object start address, and don't consider the member variables. The relatively lower coverage in Firefox is caused by its self-designed storage pool used to initialize objects. While the objects are directly initialized by `memcpy()` through the pool without calling memory allocation functions, Bitype cannot recognize them due to the mismatch of allocation patterns. However, handling these missing objects is challenging because hooking `memcpy()` may significantly raise performance overhead.

	TypeSan	HexType	Bitype		
	%	%	%	$\times_1$	$\times_2$
dealII	71.72	26.23	4.71	15.2	5.6
xalancbmk	35.63	12.51	0.78	45.7	16.0
omnetpp	49.41	16.56	8.23	6.0	2.0
soplex	4.33	-1.01	-2.68	5.3	2.7
astar	0.40	0.48	0.40	1	1.2
namd	0.23	0.16	0.16	1.4	1
povray	23.53	4.82	0.86	27.4	5.6
average	26.46	8.54	1.78	14.9	4.8
ff-octane	18.60	26.64	13.17	1.4	2.0
ff-drom-js	12.40	29.82	4.56	2.7	6.5
ff-drom-dom	71.20	129.26	30.51	2.3	4.2
average	34.07	61.91	16.08	2.1	3.9

Table 3: SPEC CPU2006 and browser benchmark performance overhead for TypeSan, HexType and Bitype. The $\times_1$ column denotes the ratio between TypeSan and Bitype and $\times_2$ denotes between HexType and Bitype.

	binary size			resident set size		
	base	Bitype	inc%	base	Bitype	inc%
dealII	3.9	4.9	25.6	821.4	1953.6	137.8
xalancbmk	5.6	7.8	71.8	423.3	815.2	92.6
omnetpp	0.8	1.1	37.5	171.6	489.8	185.4
soplex	0.5	0.6	20.0	428.1	580.3	35.5
astar	0.1	0.2	100.0	326.4	327.5	0.3
namd	0.3	0.4	33.3	48.2	49.1	1.9
povray	1.2	1.4	16.7	7.9	8.2	3.8
average			43.6			65.3
ff-octane	150.4	177.7	18.2	832.8	1782.6	114.0
ff-drom-js	150.4	177.7	18.2	503.6	1594.2	216.9
ff-drom-dom	150.4	177.7	18.2	5317.3	7996.8	50.4
average			18.2			127.1

Table 4: Memory usage for the SPEC CPU2006 C++ benchmarks and Firefox(MB)

5.2 Performance Overhead

We evaluate the performance overhead brought by Bitype according to the runtime speed. As is shown in Table 3, Bitype performs better than both TypeSan and HexType in all benchmarks, expect an equal to the astar of TypeSan. As to the performance overhead of xalancbmk, Bitype is even 45.7 times less than TypeSan, and 16 times less than HexType.

The overall low performance overhead is contributed to three reasons. Firstly, we implement the two-level lookup table to trace objects which only need three memory reads, increasing the efficiency. Secondly, we optimize the process of tracing objects by only tracing cast-related classes, which massively reduces the number of traced objects. Thirdly, our Safe Encoding Scheme allows us to do only one `xor` operation to verify typecastings.

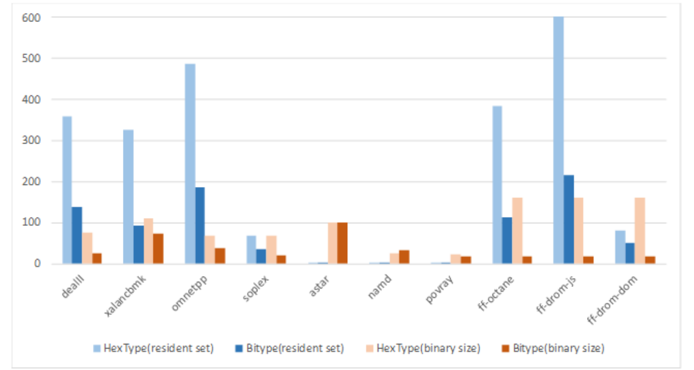


Figure 9: Comparation between HexType and Bitype in binary size overhead and resident set size overhead.

5.3 Memory Overhead

Table 4 reports the memory usage from the aspects of the static binary size and the maximum resident set size at runtime. Since TypeSan uses tcmalloc which is different from the normal allocator used by HexType and Bitype, it has no comparability and we do not compare its work with ours.

The two-level lookup table implemented in Bitype is memory mapped. Those mapped pages are not touched until they are needed (and thus actually allocated by the OS), so we only pay the memory overhead for touched areas of instantiated tables. Memory overhead of Bitype is much better than it of HexType, shown in Figure 9.

As to the maximum resident set size, Bitype has the average overhead of 65.3% for SPEC CPU2006 benchmarks, 127.1% for Firefox, while HexType's average is 176.8% and 394.1% respectively. A probable reason why HexType owns such high memory overhead is that it allocates a rather huge hash table to mitigate hash collision and reduce performance overhead as far as possible. The program astar, namd, and povray in SPEC CPU2006 are influenced slightly because they have no cast-related classes.

As to the binary size, Bitype also performs better than HexType. In SPEC CPU2006, Bitype's average overhead is 43.6% while HexType's is 67.4%. And in Firefox benchmarks, the overhead of 159.8% for HexType is much higher than our 18.2%. In general, the state-of-the-art work saves the Safe Set as a 64-bit hash value. If a Safe Set contains lots of classes, it will cost more spaces for storage, which turns a more serious problem in large-scale projects. Fortunately, our unique Safe Encoding Scheme helps us mitigate this problem, and significantly reduces the binary size compared with HexType.

5.4 Compilation Time Overhead

Table 5 shows the compilation overhead for TypeSan, HexType, and Bitype. The evaluation of Bitype concerns two conditions: the compilation including preprocessing, and compilation only. In both conditions, the overhead of Bitype is much lower than it of other two works. The compilation overhead of Bitype is 10.6 to 356.4 less times than TypeSan, and 21.8 to 223.9 times less than HexType.

Since Bitype uses the Clang Tool to collect inheritance relationships of the whole projects in advance, it avoids the duplicated

	TypeSan	HexType	Bitype(Pre+Com)	Bitype(Com)		
	%	%	%	%	$\times_1$	$\times_2$
dealII	742.7	1186.7	36.2	5.3	140.1	223.9
xalancbmk	1123.5	1060.9	142.2	25.5	44.0	41.6
omnetpp	2716.2	1912.7	140.1	53.3	50.9	35.9
soplex	207.1	1093.3	157.1	17.8	11.6	61.4
astar	251.2	513.4	163.8	23.6	10.6	21.8
namd	750.0	716.7	33.3	16.7	44.9	42.9
povray	238.9	322.2	88.9	12.7	18.8	25.4
firefox	10976.9	1392.3	184.6	30.8	356.4	45.2
average	2123.6	1024.8	118.3	23.2	91.5	44.2

Table 5: SPEC CPU2006 and Firefox compilation overhead for TypeSan, HexType and Bitype. The $\times_1$ column denoted the ratio between TypeSan and Bitype and $\times_2$ denotes between HexType and Bitype. The Bitype(Pre+Com) denotes Bitype preprocess(collect class information) time and compilation time. The Bitype(Com) denotes Bitype compilation time.

work of collection effectively and cuts down the compilation time massively.

6 DISCUSSION

Increasing Coverage. Some projects, such as Firefox, usually utilize a storage pool which defined by themselves to initialize objects. Since these initializations don't call memory allocation functions, Bitype cannot recognize them so that it cannot trace objects. Thus, here occurs the limited coverage of detection in such projects. Furthermore, different projects may be initialized by different operations, which means there may exist no particular modes to identify specific initialization. Thus, in the future work, we may possibly draw support on type system (quala[32]) to mark those initialization operations, and help Bitype to recognize them.

Fuzzing for Type Confusion. ASan[35] is a checker for some memory safety violation, and has been equipped on AFL[41] which is a frequently-used security-oriented fuzzer. Bitype is designed to check typecastings at run time, and is possible to be deployed in AFL. As a supplement to functions, in future work, our Bitype can also be integrated in AFL like ASan, to focus on the trigger and detection of new type confusion vulnerabilities.

7 RELATED WORK

The previous research works to detect type confusion are mainly UBSan[29], CaVer[25], TypeSan[19] and HexType[22] in the range of our knowledge. UBSan instruments static casts to turn them into dynamic casts, identifies objects through existing fields embedded in the objects. It does not support non-polymorphic objects and needs to manually blacklist them. CaVer is then introduced as the first typecasting verification tool. It leverages disjoint metadata to non-polymorphic classes, and traces global and stack objects by red-black trees. However, both of them suffer from prohibitively high overhead and low coverage.

The relatively new works, TypeSan and HexType, increase the efficiency of detection. TypeSan relies on a per-object metadata storage service based on a compact memory shadowing scheme, while HexType employs a hash table to reduce tracing overhead. However, to avoid the hash collision as far as possible, HexType expends the size of hash table and increases the memory overhead. We refer to the two-level lookup table mentioned in CFIXX[6], which is similar to TypeSan, and design our new Safe Encoding Scheme, leading to fact that Bitype decreases the memory overhead by 2.8 to 3.1 times to HexType on average. The elaborated Safe Encoding Scheme, moreover, makes a great contribution to the 2.1 to 14.9 times reduction in performance overhead on average. In addition, the Clang Tool[37] designed in Bitype significantly reduces the compilation time by 10.6 to 356.4 times compared to TypeSan and HexType on average.

[2, 5, 9, 15, 39, 40, 44] introduce several Control-Flow Integrity (CFI) techniques which check control-flow transfers are valid or not in the program. To a certain extent, these works can deal with the type confusion caused by control-flow hijacking. Code Pointer Integrity (CPI) [7, 24] is another kind of control-flow hijacking mitigation work. As CPI ensures the integrity of the all the code pointers in the program, it only defenses the corrupted pointers but not all invalid typecastings. Similarly, [4, 6, 21, 34, 42, 43] aim to protect virtual calls from vtable hijacking attacks and partially address the related type confusion bugs.

Some works [10, 11, 20, 23, 26–28, 35] aiming at detecting memory corruption can also find some vulnerabilities resulting from bad typecastings. As a bad typecasting may violate the neighbor memory area of the cast object, such out-of-bounds access attacks will probably lead to memory corruption and be detected. In a word, all these work cannot handle type confusion thoroughly, and inspire the design of Bitype.

8 CONCLUSION

Type confusion vulnerabilities are widely spread in popular application, and can lead to serious consequences like memory area violation and control-flow hijacking. Although more and more researchers have realized the risks brought by them, the current solutions either partially address the problem, or have high performance and memory overhead, and compilation time.

We present Bitype as an explicit typecasting checker with high efficiency and low costs based on Clang/LLVM. Bitype mainly elaborates a Safe Encoding Scheme mapping classes to bits, helps to accelerate the process of judging illegal downcastings from a source type to a target type, and decreases both performance and memory overhead. In addition, the Clang Tool for entire inheritance relationship collection, and the two-level lookup table for object tracing, and other optimization methods, limit the compilation time and memory overhead. Testing on SPEC CPU2006 benchmarks and Firefox browser, we evaluate Bitype comprehensively. While maintaining the checking coverage which is slightly higher than that of HexType, Bitype significantly reduces the performance overhead by 2 to 16 times, the memory overhead by 2 to 3 times, the compilation time by 20 to 223 times. The open-source version of Bitype is available at https://github.com/bin2415/Bitype.

ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers for their useful comments and suggestions. We would like to thank our labmates Jian Guo, Jianhao Xu and Zhilong Wang for their suggestions. This work was supported in part by grants from the Chinese National Natural Science Foundation(NSFC 61272078, NSFC 61073027).

REFERENCES

[1] 2018. Common Vulnerabilities and Exposures. https://cve.mitre.org/. Online; accessed on June 14, 2018.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[3] LLVM admin team. 2018. The LLVM Compiler Infrastructure. https://llvm.org/. Online; accessed June 3, 2018.

[4] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving.. In *NDSS*.

[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. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.

[6] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *Proceedings of the 2018 Network and Distributed System Security Symposium*.

[7] Scott A Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 193–204.

[8] Standard Performance Evaluation Corporation. 2018. SPEC CPU2006. http://www.spec.org/cpu2006. Online; accessed June 3, 2018.

[9] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 292–307.

[10] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 351–366.

[11] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack bounds protection with low fat pointers. In *Symposium on Network and Distributed System Security*.

[12] Jason Evans. 2018. Jemalloc : General Purpose Memory Allocation Functions. http://jemalloc.net/. Online; accessed on June 3, 2018.

[13] The Mozilla Foundation. 2018. DROMEAO, JavaScript Performance Testing. http://dromaeo.com/. Online; accessed June 3, 2018.

[14] The Mozilla Foundation. 2018. Mozilla Firefox. https://www.mozilla.org/en-US/firefox/53.0.3/releasenotes/. Online; accessed June 3, 2018.

[15] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 179–194.

[16] Wolfram Gloger. 2018. Ptmalloc3 : A Multi-Thread Malloc Implementation. http://www.malloc.de/malloc/ptmalloc3.tar.gz. Online; accessed on June 3, 2018.

[17] Google. 2018. Octane Benchmark. https://chromium.github.io/octane/. Online; accessed June 3, 2018.

[18] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 341–350.

[19] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 517–528.

[20] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer.

[21] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*.

[22] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. (2017).

[23] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[24] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI*, Vol. 14. 00000.

[25] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security Symposium*. 81–96.

[26] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[27] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[29] Google Chromium Project. 2018. Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Online; accessed June 3, 2018.

[30] LLVM Project. 2018. compiler-rt runtime libraries. https://compiler-rt.llvm.org/. Online; accessed June 3, 2018.

[31] LLVM Project. 2018. Writing an LLVM Pass. http://llvm.org/docs/WritingAnLLVMPass.html. Online; accessed June 3, 2018.

[32] Adrian Sampson. 2018. Quala : Type Qualifiers for LLVM/Clang. https://github.com/sampsyo/quala. Online; accessed on June 3, 2018.

[33] Paul Menage Sanjay Ghemawat. 2018. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. Online; accessed on June 3, 2018.

[34] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 448–459.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*. 309–318.

[36] Clang Team. 2018. Clang 7.0 documentation. https://clang.llvm.org/docs/ReleaseNotes.html. Online; accessed June 3, 2018.

[37] Clang Team. 2018. Clang Tool documentation. https://clang.llvm.org/docs/ClangTools.html. Online; accessed June 3, 2018.

[38] Clang team. 2018. Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Online; accessed June 3, 2018.

[39] 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 *USENIX Security Symposium*. 941–955.

[40] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 927–940.

[41] M. Zalewski. 2018. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Online; accessed on June 14, 2018.

[42] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *NDSS*.

[43] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *NDSS*.

[44] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*. 337–352.