Enforcing C/C++ Type and Scope at Runtime for Control-Flow and Data-Flow Integrity

Mohannad Ismail
Virginia Tech
Blacksburg, Virginia, USA

Christopher Jelesnianski
Virginia Tech
Blacksburg, Virginia, USA

Yeongjin Jang
Samsung Research America
Mountain View, California, USA

Changwoo Min
Igalia
Seoul, South Korea

Wenjie Xiong
Virginia Tech
Blacksburg, Virginia, USA

Abstract

Control-flow hijacking and data-oriented attacks are becoming more sophisticated. These attacks, especially data-oriented attacks, can result in critical security threats, such as leaking an SSL key. Data-oriented attacks are hard to defend against with acceptable performance due to the sheer amount of data pointers present. The root cause of such attacks is using pointers in unintended ways; fundamentally, these attacks rely on abusing pointers to violate the original scope they were used in or the original types that they were declared as.

This paper proposes *Scope Type Integrity (STI)*, a new defense policy that enforces *all pointers (both code and data pointers)* to conform to the original programmer's intent, as well as *Runtime Scope Type Integrity (RSTI)* mechanisms to enforce STI at runtime leveraging ARM Pointer Authentication. STI gathers information about the scope, type, and permissions of pointers. This information is then leveraged by RSTI to ensure pointers are legitimately utilized at runtime. We implemented three defense mechanisms of RSTI, with varying levels of security and performance tradeoffs to showcase the versatility of RSTI. We employ these three variants on a variety of benchmarks and real-world applications for a full security and performance evaluation of these mechanisms. Our results show that they have overheads of 5.29%, 2.97%, and 11.12%, respectively.

ACM Reference Format:
Mohannad Ismail, Christopher Jelesnianski, Yeongjin Jang, Changwoo Min, and Wenjie Xiong. 2024. Enforcing C/C++ Type and Scope at Runtime for Control-Flow and Data-Flow Integrity . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651342

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651342

1 Introduction

Control-flow hijacking and data-oriented attacks have become more sophisticated in recent years. For example, attacks such as DOP [44] and NEWTON [81] exploit data pointers to leak sensitive data, such as SSL keys, or achieve arbitrary code execution. These attacks bypass many state-of-the-art defense mechanisms [6, 17, 25, 31, 35, 38, 39, 43, 50, 53, 56, 64–67, 77, 78, 80, 82, 85, 86]. Such data-oriented attacks are more difficult to defend against as data pointers are much more abundant in programs than code pointers. It is also not easy to distinguish between security-sensitive data pointers (e.g., pointing to an SSL key) and non-security-sensitive data pointers. These attacks abuse pointers in unintended ways, far from what was originally intended by the programmer.

Several defense mechanisms have been proposed to protect data pointers [21, 26, 40, 71]. However, they suffer from reliance on programmer annotation, large dynamic metadata, partial protection (protecting only a subset of data pointers), and/or high overhead. Reliance on programmer annotation [72] makes it challenging to protect legacy programs. Moreover, the programmer may not accurately annotate everything, resulting in more possible bugs. Large dynamic metadata [53] can be abused by an attacker to bypass defenses that heavily rely on metadata for enforcement. More importantly, a creative attacker may abuse other non-instrumented pointers to bypass the defense mechanism if the program is only partially instrumented.

During a control-flow hijacking or data-oriented attack, pointers are corrupted at runtime and the program behaves anomalously, *i.e.*, in a way that is not intended by the programmer. In C/C++ programs, variables have a specific type, are valid inside a specific scope, and have specific permissions. These are all restrictions imposed by the programmer. We refer to these properties as the programmer's intent. However, at runtime, these restrictions are lost and are not enforced in machine code. In a typical attack, the programmer's intent is violated and the pointers are abused. As in

prior work, our design philosophy is to leverage the restrictions inherently defined in the source code at runtime, so that the program can be executed with the proper programmer's intent, leaving little room for attackers to manipulate the program.

To this end, this paper proposes *Scope-Type Integrity (STI)*, a new defense policy that enforces pointers to conform to the programmer's intent, by utilizing scope, type, and permission information. STI collects information at compile time about the type, scope, and permission (read/write) of every pointer in the program. This information can then be used at runtime to enforce that pointers comply with their intended purpose. This allows STI to defeat advanced pointer attacks since these attacks typically violate either the scope, type, or permission.

We present *Runtime Scope-Type Integrity (RSTI)* mechanisms. RSTI leverages ARM Pointer Authentication (PA) to generate Pointer Authentication Codes (PACs), based on the information from STI, and place these PACs at the top bits of the pointer. At runtime, the PACs are then checked to ensure pointer usage complies with STI. In this way, pointers are guaranteed to execute conforming to the programmer's intent expressed in the application. This allows integrity checks to be performed without needing much external metadata and provides high-security guarantees against creative attacks. Also, leveraging ARM PA allows runtime checks to be efficient, thus opening the way to efficiently and securely protect all pointers in a program. We introduce RSTI-STWC (Scope-Type without Combining), which is our main RSTI mechanism. RSTI-STWC protects all pointers (both code and data pointers) in a program and re-signs pointers whenever a cast happens so that the type semantics of the program can still be conformed to. We further introduce two variants of RSTI-STWC: RSTI-STC (Scope-Type with Combining), a relaxed version of RSTI-STWC, and RSTI-STL (Scope-Type with Location), a stricter version of RSTI-STWC. With these three mechanisms, we are able to show the variety of trade-offs between security and performance of RSTI. We also describe the precision of the enforcement of each mechanism and the security implications of the design decisions of each mechanism. We use the term *Equivalence Class* to quantify the protection precision. *Equivalence Class* here measures how unique each pointer type or variable is in the program, and this quantifies how viable pointer substitution attacks can be within an application. To this end, we make the following contributions:

- We introduce a new defense policy, Scope-Type Integrity (STI), that enforces pointers to conform to the programmer's intent. It leverages scope, type, and permission information expressed in the source code to be used after it has been lost during compilation. To our knowledge, STI is the first of its kind to defend against pointer-based attacks by hardening the programmer's intent into machine code.
- We propose Runtime Scope-Type Integrity (RSTI), efficient enforcement of STI using ARM's Pointer Authentication (PA). We introduce three RSTI mechanisms, RSTI-STWC, RSTI-STC, and RSTI-STL, that instrument all pointers (both code and data pointers) with different restrictions, showing a trade-off between security guarantees and performance overhead.
- We prototype all three RSTI mechanisms on the LLVM compiler, and demonstrate the feasibility of the proposed schemes on a commercial processor.
- We provide a comprehensive security evaluation of RSTI and its mechanisms on state-of-the-art control-flow hijacking and data-oriented attacks, and real-world CVEs.
- We evaluate RSTI's mechanisms on a variety of benchmarks and real-world programs, including SPEC CPU 2006, SPEC CPU 2017, nbench, NGINX and CPython PyTorch, with average overheads of 5.29%, 2.97%, and 11.12% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

2 Background and Motivation

2.1 Control-flow Hijacking

Control-flow hijacking attacks are critical because they may allow attackers to run arbitrary code. A popular way to carry out a control-flow hijacking attack is exploiting memory corruption vulnerabilities, which C/C++ programs are prone to having. In particular, attackers can alter the value of a code pointer (*e.g.*, function pointers) by corrupting the memory location that stores the pointer to subvert execution flow of a program [14, 15, 20, 28, 34, 37, 74].

The control-flow hijacking attack in Figure 1 shows a vulnerability (CVE-2015-8668) in the `libtiff` library. This is

```
1  int TIFFWriteScanline(TIFF* tif, ...){
2    ...
3    // Function pointer dereference
4    // Arbitrary address
5    // Execute attack!!
6    status = (*tif->tif_encoderow)(tif, (uint8*) buf,
7                  tif->tif_scanlinesize, sample); }
8  void _TIFFSetDefaultCompressionState(TIFF* tif){
9    // Function pointer assignment
10   tif->tif_encoderow = _TIFFNoRowEncode; }
11 TIFF* TIFFOpen(...){
12   ...
13   _TIFFSetDefaultCompressionState(tif);}
14 int main(int argc, char* argv[]){
15   TIFF *out = NULL;
16   out = TIFFOpen(outfilename, "w");
17   ...
18   uint32 uncompr_size;
19   unsigned char *uncomprbuf;
20   ...
21   uncompr_size = width * length;
22   // Unsanitized Code - Buffer overflow
23   uncomprbuf = (unsigned char *)_TIFFmalloc(
24                           uncompr_size);
25   ...
26   if (TIFFWriteScanline(out, ...) < 0) {}
27   ...}
```

Figure 1. Control-flow hijacking attack example. The attacker can exploit the buffer overflow vulnerability in Line 21.

```
1 int serveconnection(int sockfd) {
2   char *ptr;
3   ...
4   if (strstr(ptr, "/..")) // Reject the request
5   log( ... ); // Buffer overflow!
6   ...
7   if (strstr(ptr, "cgi-bin") // Handle CGI request
8   ...  }
```

Figure 2. Data-oriented attack example. The attacker corrupts `ptr` in Line 4 to bypass the security checks.

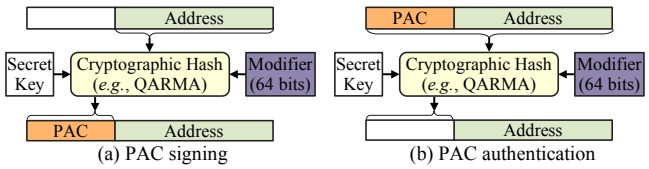


Figure 3. PA mechanism signs a pointer and produces a Pointer Authentication Code (PAC) based on the pointer, a user-provided modifier, and a secret key.

a heap-based buffer overflow. The program does not sanitize the buffer size in Line 21. This means that `uncomprbuf` can be too small, allowing the attacker to overflow heap memory. A possible target is `tif_encoderow`, which is called by `TIFFWriteScanline`. The attacker can then overwrite `tif_encoderow` with an arbitrary address that they can jump to when the function pointer gets dereferenced.

2.2 Data-Oriented Attack

Data-oriented attacks aim to corrupt non-control data pointers in order to maliciously leak information [23, 44] or achieve arbitrary code execution. These attacks are much more powerful than control-flow hijacking attacks, due to the fact that they do not touch any control data. They have been used, for example, to leak an SSL key [33, 44]. They are harder to defend against due to the abundance of data pointers in a program and the inability to distinguish security-sensitive data pointers from non-security-sensitive data pointers.

Figure 2 shows an attack against the GHTTPD web server [24]. In this attack, the attacker relies on corrupting the pointer `ptr`. They send an HTTP request with a crafted URL. Then, they trigger the buffer overflow vulnerability in `log()` and overwrite the address in pointer `ptr` to the address of the crafted URL. This crafted URL allows the attacker to bypass the input validation checks in Lines 4 and 7, and the attacker executes `/bin/sh`. Manipulating data pointers is often the desired attack vector for data-oriented attacks [26].

2.3 Scope, Type, and Permission in C/C++

When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:

- **Basic Type**: Each variable must have a specific type, *e.g.*, `char`, `int*`. This type is defined by the programmer to tell the compiler how this variable will be used in the program.
- **Scope**: Scope defines where the variable will be used. For example, in Figure 2, the pointer `ptr`'s scope is the function `serveconnection`, and should not be used outside that.
- **Permission**: We refer to permissions as whether a variable is defined as read or read/write. A programmer usually defines this by using `const` in the variable definition.

These three properties express what we refer to as the *programmer's intent* for the usage of variables. However, such information is lost after compilation. Thus, an attacker can easily weaponize pointers without conforming to their proper usage. STI aims to analyze the program and retrieve this information. Then, this information is passed to RSTI to enforce the policy at runtime with ARM PA. Note that STI only concerns itself with pointer variables, since this is the variable type that is usually manipulated by attackers.

2.4 ARM Pointer Authentication

We leverage the Pointer Authentication feature in ARM to enforce RSTI. ARM introduced Pointer Authentication (PA) in ARMv8.3-A [45] and is available in commercial machines. PA is a hardware security feature that aims to protect the integrity of pointers. It does this by generating a Pointer Authentication Code (PAC) with a cryptographic hash algorithm. For *signing*, the algorithm takes a pointer, a 64-bit modifier, and a secret key. It then generates a PAC that is placed at the top unused bits of the pointer, as shown in Figure 3(a). For *authentication*, the algorithm takes the PAC'ed pointer, as well as the same modifier and key. The PAC is then recalculated and checked with the PAC on the pointer. If they match, the PAC is removed, as shown in Figure 3(b). If they do not, the top two bits of the pointer are flipped, causing the pointer to be unusable. PA has `pac` instructions for signing pointers and `aut` instructions for authenticating.

We leverage PA in RSTI to enforce scope, type, and permission. RSTI instruments all pointer loads/stores with PA instructions (`pac`/`aut`), leveraging LLVM's pointer authentication intrinsics (`llvm.ptrauth`) [2]. `llvm.ptrauth.sign` and `llvm.ptrauth.auth` take three arguments:

- **`pointer address`**: This is the raw pointer to be signed or authenticated.
- **`key`**: This is an integer identifying which key will be used.
- **`modifier`**: a 64-bit integer that adds additional diversity to the PAC.

3 Threat Model and Assumptions

We follow a threat model of typical memory-corruption attacks [76]. The attacker's goal is to achieve arbitrary code execution or memory access by hijacking control/data flow by abusing a memory corruption vulnerability. RSTI does not prevent corruption of pointers but rather prevents an attack from achieving code execution stemming from arbitrary read/write that can result from abusing pointers. We assume that Data Execution Prevention (DEP) is in place,

and thus the attacker cannot inject their own code. DEP is now enabled by default in most modern operating systems [49, 61]. Also, we trust the hardware and the kernel; more specifically, we trust that PA keys are securely generated, managed, and stored by the kernel. In addition to that, we also assume a return address protection mechanism, such as shadow stack or an equivalent mechanism [5, 10], is implemented. Attacks that target the kernel and hardware, such as transient execution attacks [52, 68], are out of scope.

4 Runtime Scope-Type Integrity (RSTI)

4.1 Design Goals

The main goal of RSTI is to protect *all pointers* in a program from memory corruption attacks. This puts RSTI in a position similar to Data Flow Integrity (DFI) [22] and other similar techniques [26, 47, 55]. Some techniques that protect data pointers are limited by their reliance on programmer annotation [71]. Other mitigation techniques [47, 53] rely on external metadata in memory. This causes high overhead, due to the abundance of data pointers, and exposes the metadata to an attacker. Thus, in designing RSTI, we wanted to overcome these challenges. In summary, our main goals are:

- **Completeness:** Protection of all pointers (both code and data pointers) in a program.
- **Little reliance on external metadata:** Eliminate metadata lookup overhead and attacks on the metadata.
- **High performance:** Keep runtime overhead low.
- **Compatibility**: Allow protection of legacy (C/C++) programs without needing programmer annotations.

4.2 Design Philosophy

The goal of the defense mechanisms is ensuring correct and proper execution of a program. Even though a programmer puts significant effort and information into writing a program to ensure it executes in the desired way, all this information is, unfortunately, lost at runtime. So why not leverage this information as the security context, and propagate it to the runtime in some way? If we can ensure that the program executes in the way that the programmer intended even when an attacker is present, then the program would not be compromised, since an attacker relies on the anomalous execution of a program by exploiting vulnerabilities. By programmer's intent, we refer to the various programming constructs that are used by the programmer to write the program. However, these constructs need to be carefully chosen, in order to guarantee enough uniqueness for the security context between different pointers. We identified three main vital pieces of information that are defined by a programmer to execute the program correctly: *scope*, *type*, and *permission* (defined in §2.3). The anomalous execution of a program is the main primitive relied on by attacks such as Return Oriented Programming (ROP) [70]. In addition, defending against anomalous execution of programs by enforcing intent is the main goal of defense techniques such as CFI [6], CPI [53], DFI [22] and other defense mechanisms [18, 43, 50, 75]. CFI guarantees the transfer of control flow to a valid destination, represented by a Control Flow Graph (CFG) and labels. RSTI leverages the STI information for anomalous detection, and we show the practicality and effectiveness of it against a wide variety of attacks. Below, we show how to defend against a control-flow hijacking attack and data-oriented attack, in Figures 1 and 2, respectively.

Defending control-flow hijacking. For the control-flow hijacking attack in Figure 1, if we can enforce that the pointer `tif->tif_encoderow` conforms to its type (`TIFFCodeMethod`) and its scope (`TIFFWriteScanline` and `_TIFFSetDefaultCompressionState`), then the attacker would not be able to overwrite it with an arbitrary pointer. The attacker wouldn't even be able to use another valid pointer in the program if it does not meet these restrictions.

Defending data-oriented attacks. By enforcing scope, type, and permission, the attacker cannot easily overwrite the data pointer `ptr` in Figure 2. The attacker can only corrupt the pointer with an alternative pointer of the same type (`char*`) and the same scope (`serveconnection`). This reduces the attack vector significantly.

4.3 Design Overview

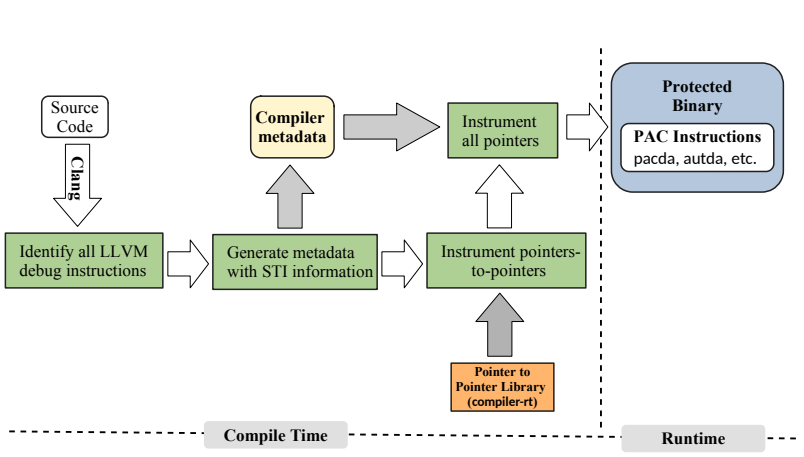
RSTI aims to protect all pointers from being abused by enforcing Scope-Type Integrity (STI) at runtime. STI ensures that pointers are dereferenced from the correct scope, with the correct type and abide by the correct permissions. If an attacker corrupts a pointer, they cannot manipulate the PAC to bypass authentication. This allows RSTI to enforce the programmer's intent without relying on extra annotation.

Figure 4 shows the overall design of RSTI. At compile time, all pointers in the source code are instrumented depending on which RSTI mechanism is chosen. The RSTI compiler collects and analyzes all the LLVM debug information to generate compiler metadata. This metadata is then used by LLVM `ptrauth` intrinsics to generate the protected binary with PAC instructions. At runtime, ARM PA is used to efficiently enforce STI.

4.4 Scope, Type, and Permission

Programmer's intent on pointer properties. STI uses type, scope, and permission information as the programmer's intent to enforce protection. Such information can be collected at compile time. We show that STI is secure enough to ensure that a program executes as the programmer intended in §6.1.

- **Basic Type**: Enforcing the type at runtime allows the program to interpret a raw address in the intended way. This is a useful security context since many attackers rely on type-confusion. Type-confusion here means *illegal* type




```
1  // C code
2  int main(void){
3          const void *cp = malloc (sizeof(void));
4  }
1  //LLVM IR
2  %2 = alloca i32*, align 8
3  call void @llvm.dbg.declare(
4                  metadata i32** %2, // Type
5                  metadata !13, !dbg !16
6  %3 = call i8* @malloc(i64 1) #3, !dbg !17
7  %4 = bitcast i8* %3 to i32*, !dbg !17
8  store i32* %4, i32** %2, align 8, !dbg !16
1  // LLVM IR debug info
2  !9 = distinct !DISubprogram(name: "main", scope: !6,
3          file: !6, line: 14, type: !10, scopeLine: 15
4  !13 = !DILocalVariable(name: "cp", scope: !9,
5          file: !6, line: 16, type: !14) // Scope
6  !14 = !DIDerivedType(tag: DW_TAG_pointer_type,
7          baseType: !15, size: 64)
8  !15 = !DIDerivedType(tag: DW_TAG_const_type,
9          baseType: !12) // Permission
10 !16 = !DILocation(line: 16, column: 13, scope: !9)
```

Figure 4. Overview of RSTI. Starting with the source code, RSTI compiler identifies the scope-type information, generates the metadata, and instruments all pointers. The metadata is static and only used during the compilation phase. The code snippet on the right shows how the scope, type, and permission information is identified by RSTI through the `llvm.dbg` information.

confusion, in which an attacker replaces a pointer of one type with a pointer of another *incorrect* type. By *incorrect* type, we mean against the programmer's intent. If a cast is in the code, it is considered correct from RSTI 's perspective.

- **Scope**: Enforcing the scope at runtime allows the use of a pointer variable to be bound to a certain subset region of program code. Our definition of scope extends the "scope" in the C language to also consider composite types. In the case of function scopes, it is the set of functions that use the variable. For example, a programmer may define a variable `void* p` to be used in a function `foo()`. Thus, the scope of `p` is `foo`. This specific example is for a local variable. If the variable escapes, *i.e.*, it is used in other functions, then the scope is widened to these other functions. If there is a compound statement in a function, that does not constitute a new scope. In the case of composite types, such as structures, the structure itself is included in the scope. For example, if a `struct bar` has a member variable `void* a`, then `bar` is also included in the scope of `a`. In the case of nested structs, the scope is assigned iteratively for the pointers to the variable. For example, if a `struct foo` contains a `struct bar` and `struct bar` has a variable `void* a`, then the scope of `a` would only include `struct bar`, and the scope of the pointer to `struct bar` would include `struct foo`.
- **Permission**: Enforcing the permission at runtime allows read-only variables to not be abused by an attacker with read/write variables.

Extracting scope, type, and permission from LLVM IR. LLVM's Intermediate Representation (IR) allows us to obtain all of the scope-type information with LLVM's IR metadata. LLVM generates `llvm.dbg` instructions, that can be used to access the metadata of the variable. STI uses these instructions to initialize the internal static metadata with the scope, type, and permission information. Figure 4 (right) shows a code snippet from a C program and the corresponding LLVM debug metadata. The variable `cp` is defined with a type of `void*`. Its scope is the function `main`. It has read-only permissions, due to being `const`. The type information is in Line 4 of the LLVM IR, which is the `i32*` type. The scope information is initially obtained from the instruction `!13` by traversing the `llvm.dbg` instruction, to get to the `!DILocalVariable`. When instrumenting loads/stores, the scope is obtained with the `!16` instruction and every load/store has this LLVM metadata. Thus, this means the proper scope can always be obtained. Permission information requires further traversal of the `!DILocalVariable` to reach `!DIDerivedType`. There are multiple layers of `!DIDerivedType` metadata and we check their tags to find the `DW_TAG_const_type` tag. This specific tag denotes that this variable is read-only. Scope-type information is then stored in internal compile-time metadata.

Pointer Casting. C/C++ semantics allow types to be cast from one type to another. Thus, those two types become *compatible*, since it is explicitly done by the programmer or by the compiler. If a pointer gets cast from one type to another, then we define these types as *compatible types*.

4.5 RSTI-types

We define *RSTI-type* as a pointer type which restricts a pointer to conform to the scope-type information. If a pointer does not have the intended RSTI-type, RSTI will detect that the pointer has been tampered with. For example, the table in Figure 5a shows two RSTI-types (M2 and M3) for one basic type (`void*`), due to the fact that there are two `void*` variables that have different scopes and different permissions.

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

(a) RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M2);
19
20
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*, void*	main,foo2	R/W	M1
void*	main	R	M2

(b) RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const, 2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   foo2((void*) c);
22   ...}
```

Type	Scope	Permission	RSTI Type
ctx*	main	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3
ctx*	foo	R/W	M4
ctx*	bar	R/W	M5

(c) RSTI-STL

Figure 5. Code examples with instrumentation. The table below each snippet shows how the RSTI-type is internally stored.

Distinguishing between local and escaping variables. RSTI-type also distinguishes between variables that are local to a function versus variables that escape. The local variable's scope would only have the one function it is in, compared to an escaping variable's scope would include all functions it is used in, *e.g.*, M1 in the table in Figure 5a. The function `isNonEscapingLocalObject` in LLVM helps RSTI with this distinction. This allows RSTI to be precise in its enforcement, whilst maintaining correctness of the program. We clarify the RSTI-type with examples and explain the different RSTI defense mechanisms in the next section.

4.6 Enforcement and Defense Mechanisms

RSTI instruments all pointer load/store instructions and enforces RSTI-type with PAC. RSTI instruments all pointers on the stack and heap, including globals. RSTI decides the scope based on where that pointer is legitimately defined and used in the program. The RSTI-type is included in the modifier that is passed to the cryptographic algorithm. Any change in any one of the components means that a different PAC will be generated, and thus will fail on authentication. This mechanism allows STI's restrictions to be enforced at runtime. RSTI supports uninstrumented libraries by stripping the PAC when an external library call is made.

We propose three RSTI defense mechanisms, each with its own distinct view of the RSTI-type. RSTI's three security mechanisms have different security guarantees due to the different nature and strictness each mechanism has to offer.

RSTI-STWC (Scope-Type Without Combining) is our main RSTI mechanism. RSTI-STWC authenticates and re-signs pointers when casts happen. For example, in Figure 5a, there are two basic types in the program (`void*` and `ctx*`). Due to RSTI-STWC's analysis considering scope and permission, there are now three *RSTI-types*. Even though there is a cast between `void*` and `ctx*`, RSTI-STWC maintains separate RSTI-types for them. RSTI-STWC makes sure that legitimate casts are handled by re-signing the PAC with the RSTI-types after casting, as in Lines 19-21 in Figure 5a. However, RSTI-STWC would not be able to detect if two pointers with the same scope-type information are substituted.

RSTI-STC (Scope-Type with Combining) combines compatible types together so that it does not need to re-sign pointers when pointer casts happen. For example, in Figure 5b, `ctx*` and `void*` are combined into *one RSTI-type*, and thus there is no need for additional instrumentation to handle them. The viability of a pointer substitution attack depends on whether the scope-type information of the pointers being substituted are both compatible or not. If they are not compatible, then RSTI-STC can detect the attack. If they are, then RSTI-STC falls short here. RSTI-STC has less security guarantees but offers better performance. This is discussed in detail in §6.

RSTI-STL (Scope-Type with Location) is the strictest RSTI defense mechanism. It combines the location, *i.e.*, address, of the pointer with the scope-type information to completely prevent any pointer substitution attacks. In addition to instrumenting casts, RSTI-STL re-signs any pointers

that get passed as arguments, due to a change in the location of the pointer. RSTI-STL includes the location of a pointer p (&p) in the modifier. This means that the pointer can only be used legitimately at that specific location [47]. Anytime the location of the pointer changes, RSTI-STL authenticates and re-signs within the new location. This allows it to overcome the shortcomings of the others, albeit at a higher performance cost due to a higher number of instrumentations.

Precision of the protection A replay attack is where an attacker maliciously reuses pointers with the same PAC in a different context. As can be seen in Figure 5, each mechanism enforces different security guarantees against this attack. This means that the precision of enforcement of each mechanism is different. For example, since RSTI-STC combines compatible RSTI-types, more pointers would have the same RSTI-type. This increases the likelihood of replay attacks. However, RSTI-STWC and RSTI-STL do not combine RSTI-types and have higher precision than RSTI-STC. PARTS [55] uses the pointer's basic type as the modifier, which can be reused maliciously by an attacker [47]. RSTI's mechanisms rely on more than the basic type and thus have more precision in general than PARTS. We elaborate on this in §6.2.1.

4.7 Enforcement Details

4.7.1 On-Store Pointer Signing. RSTI instruments all pointer stores in a program. They are all signed with their respective RSTI-type as the modifier. Thus, all pointers in a program always have a PAC on them and are always protected. For example, Line 14 in Figure 5a.

4.7.2 On-Load Pointer Authentication. RSTI authenticates pointers as they are loaded from memory, using the same RSTI-type that was used to sign them on-store. The LLVM pointer authentication intrinsics allow authentication to happen without spilling to memory, due to them being optimized in the compiler. This means that we do not need to re-sign pointers after authentication, since the compiler optimizes the memory accesses and the authenticated address is always in a register. For example, Line 19 in Figure 5a.

4.7.3 Pointer Operations. Due to the fact that PAC changes pointer semantics, care must be taken when instrumenting pointers in a program.

Universal pointer types. Universal pointer types such as `void*` and `char*` are abundant in C/C++ programs. RSTI treats them just as it would treat any other type. This allows for consistent instrumentation across programs that may have their own types that are abundant as well.

Pointer Arithmetic. RSTI supports pointer arithmetic operations. However, RSTI-STWC and RSTI-STC are not capable of enforcing bounds on a pointer. RSTI-STL adds the location of the pointer to the modifier and is able to enforce correct pointer arithmetic without needing bounds. The location of the pointer is determined by the address of the pointer, and this value doesn't change even when the pointer is incremented or decremented.

```
1  void hello_func(){printf("Hello!"); }
2  struct node {
3     int key;
4     int (*fp)();
5     struct node *next; };
6  int main(void){
7      struct node* ptr = (struct node*)
8                           malloc(sizeof(struct node));
9      ptr->fp = hello_func;
10     ptr->fp(); }
```

Figure 6. Composite type example. RSTI handles composite types and enforces the scope of its members to that type.

4.7.4 Field Sensitive Analysis. RSTI does field-sensitive analysis on composite type variables in order to achieve finer-grained and accurate enforcement of the programmer's intent. RSTI handles members of a composite type and assigns the scope and type appropriately. Figure 6 shows an example of a variable `ptr`, which is of type `struct node*` and its scope is `main`. Its member variable, `fp`, has a type of `int *()` but its scope is both `main` as well as `struct node`. RSTI enforces that the `ptr->fp()` has the proper scope in terms of the functions it is executed in, as well as its composite type. Note that composite types can act as both scope and type, depending on which pointers are being referred.

As for composite types, LLVM identifies them in the debug metadata. Every struct type has a `!DICompositeType` in the LLVM IR. This allows RSTI to distinguish composite types from other types that use the `!DIDerivedType`. Enforcing the scope and type on struct pointer members is twofold. First is using the actual type the member is defined in. The actual type is represented in IR. The second is enforcing the struct type where the pointer is a member. This is done by accessing the GEP (`getelementptr`) instruction. Struct members are accessed in IR with the GEP instruction, and this can be used to get the struct type and reach the `!DICompositeType` to enforce the struct type.

4.7.5 Type Punning and Inheritance. Type punning is a form of pointer aliasing in which two pointers refer to the same location in memory but they represent that location as different types. One of the ways in which this is done in C code is through casting. RSTI handles type punning to make sure that each pointer has a PAC corresponding to its correct type. This is done by detecting the `BitCast` instruction in the LLVM IR, and subsequently finding the corresponding RSTI-type for the variable. The pointers are authenticated and re-signed in the case of RSTI-STWC, and RSTI-types are combined in the case of RSTI-STC. Inheritance in C++ is where a base class pointer can be used to access objects in a child class. The base class and child class are considered to be of different types. Even though there are no explicit casts being done in the code, the LLVM IR emits `BitCast` instructions whenever a base class accesses functions in the

child class. This allows inheritance to be handled in a manner similar to type punning.

4.7.6 Protection of Heap Variables. RSTI's compile-time analysis extracts the scope, type, and permission for all pointers and instruments all the pointer load/store instructions on both the stack and *heap*. From the IR's perspective, heap access is just another memory access, and the pointers that access the heap objects have their scope-type information in the IR. RSTI does not track the lifetime of the pointer. RSTI utilizes the scope-type information to make sure that this heap object is accessed by a legitimate function, in a legitimate scope, and with legitimate permissions. Accessing the heap is done in one of three ways: through a global variable, a local variable, or a pointer embedded in the heap object. Our field-sensitive analysis, explained in §4.7.4, accommodates these and showcases how RSTI utilizes the scope-type information to guarantee correct access. An example of that is in Figure 6, where `ptr` is a pointer of type `struct node` that points to a heap object, and its scope is the function `main`. Its member variable, `fp`, has a type of `int *()`. However, its scope is both `main` and `struct node`. RSTI enforces that `ptr->fp()` has the proper scope in terms of the functions it is executed in, as well as its composite type. This field-sensitive analysis allows the embedded pointer in a struct to have the proper scope-type information.

4.7.7 Pointer-to-Pointer handling. Due to reliance on the RSTI-type when signing/authenticating pointers, we must make sure that the correct types are being used and propagated. For single pointers, this is not an issue. However, for a pointer-to-pointer, this can be an issue. When a double pointer is type-casted and passed as a function argument, the original type of the pointer can be lost. Thus, we need to find a way to preserve the original type of the pointer in order to make sure that it does not violate the programmer's intent. Our solution is to store the original type on the pointer itself. However, the number of bits on the pointer that are available (8 bits in case of ARMv8) is limited. We resolve this by adding an extra step of indirection. We define a tag that gets added to the top bits and that leads us to the full original type along with its modifier from the RSTI-type. The tag and the full original type are referred to as the Compact Equivalent (CE) and Full Equivalent (FE), respectively.

Figure 7 shows how the mechanism works. Functions `foo1` and `foo2` take `&p`, a pointer-to-pointer, as an argument. For `foo1`, the same basic type (`struct node`) is in the argument and thus the pointer-to-pointer mechanism is not needed. However, for `foo2`, the type is `void**`. In this case, the original type (`struct node**`) is lost and thus the mechanism is needed. Thus, our pointer-to-pointer solution resolves this by storing a tag (CE) that refers to the original type (FE) on the top 8 bits of the pointer. The CE and FE are also both stored in *read-only metadata* in memory and can only be accessed by the pointer-to-pointer library in `compiler-rt`.

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                       malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```

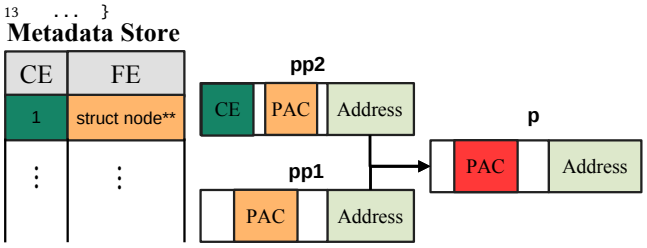


Figure 7. Pointer-to-pointer handling to preserve the original type. The Compact Equivalent (CE) and Full Equivalent (FE) refer to a tag and the original type of the pointer-to-pointer, respectively.

To implement this, RSTI makes use of an ARMv8 hardware feature called Top Byte Ignore (TBI) [12]. TBI is a feature that ignores the top 8 bits of a virtual address.

Usage. Not all pointer-to-pointer types need to be covered. Only ones that are lost when the pointer type goes out of scope, (*e.g.*, being cast and passed as a function argument) and thus cannot be statically detected. While there are different pointer-to-pointer scenarios, few pointer-to-pointers fall into this specific category. The IR provides sufficient information to handle pointer-to-pointer cases that do not fall into this category. Thus, we do not need to instrument every pointer-to-pointer with this mechanism. Since the CE can only be 8 bits, this means that only 256 types can be used. We evaluate the usage of our mechanism in §6.2.2.

Enforcement. RSTI checks for all instances where a pointer-to-pointer is being cast and then passed as a function argument. Then, the CE and FE are obtained, added to the table, and the pointer is signed with a PAC based on its RSTI-type and based on the CE. When authenticating a pointer-to-pointer, RSTI checks to see if that RSTI-type exists. Then, RSTI uses the CE to obtain the modifier for the original type from the table in memory. In this way, the original type of the pointer can be obtained and authentication can be done.

The main goal of the pointer-to-pointer mechanism is to preserve the original type the pointer was being casted from, and thus the mechanism can support any level of indirection. So, for example, if a `struct node***` pointer was cast to a `void*` and passed as a function argument, the FE would store the original type as `struct node***` in the metadata and place the corresponding CE on the pointer. Thus, the original type can always be inferred regardless of the casted type. The same thing would happen if a `struct node**` was cast to a `void*`. If a `struct node**` was cast to a `void*` and

then stored in another struct, the pointer will be authenticated and re-signed with the proper struct composite type, due to the existence of a `BitCast` instruction and a `store` instruction. We have not had any issues pertaining to this in our evaluation. RSTI uses a runtime library to handle pointer-to-pointer. Its functions are:

- **`pp_add`**: This is called when a casted double pointer function argument is detected. It adds the FE to the metadata along with its modifier from the internal compiler metadata (Line 9 in Figure 7).
- **`pp_sign`**: This is called before a store of a casted double pointer function argument. It signs the pointer with a PAC based on the RSTI-type and metadata (Line 10 in Figure 7).
- **`pp_auth`**: This is called before a load of a signed casted double pointer. It authenticates that pointer with the RSTI-type and the original type obtained from the metadata (Line 3 in Figure 7).
- **`pp_add_tbi`**: This is called after `pp_sign`. It adds the CE to the top bits of the pointer so that the the original type can be obtained when `pp_auth` is called (Line 11 in Figure 7).

4.8 Merging of Compatible Types for Casting

Depending on which mechanism is used and the pointer casts, different RSTI types can be combined or merged together. This is showcased in Figure 8. There are two basic types, `void*` and `int*`, but each RSTI mechanism distinguishes between them depending on the code. RSTI-STC merges types if there is a cast (Line 5), and thus both basic types would have one RSTI-type under RSTI-STC. RSTI-STWC doesn't merge types with casts. However, since `p1` and `p2` have the same scope, type, and permission, they both will have one RSTI-type. RSTI-STL adds the location, and this allows it to distinguish `p1`, `p2`, and `p3` with three RSTI-types. This distinction directly affects the number of instrumentations for each mechanism.

5 Implementation

Our prototype is built on top of Apple's LLVM fork [1]. It consists of 3,504 lines of code (LoC), with 3,017 LoC for the LLVM pass and 487 LoC for the pointer-to-pointer library. This includes all three mechanisms. The pointer-to-pointer library is integrated into LLVM's `compiler-rt`. The LLVM pass executes on the IR level but is in the AArch64 backend. RSTI is available at: https://github.com/cosmoss-jigu/rsti.

We apply a few optimizations to RSTI. First, is Link Time Optimization (LTO), which combines all the object files into one file. Also, we inline all the pointer-to-pointer library functions and we execute the pass in the LTO phase. This allows RSTI to avoid unnecessary instrumentation.

Executing the pass in the LTO phase, particularly after all the object files have been combined into one, is not only beneficial for performance reasons, but is important to help RSTI function properly and efficiently. Being able to have a full look at the program and analyze all the scope-type information at once, instead of doing it per object file, allows STI to completely map all the scope-type information for all the pointers accurately for the entire program.

```
1 void foo(){
2    void *p1, *p2;
3    int* p3;
4    ...
5    p1 = (void*) p3;
6    ...}
```

	RSTI-STWC	RSTI-STC	RSTI-STL
p1 and p2	Merges RSTI-type of p1 and p2	Merges RSTI-type of p1 and p2	Doesn't merge
p1 and p3	Doesn't merge	Merges RSTI-type of p3 with p1 and p2	Doesn't merge

Figure 8. RSTI merging of types. The variation allows for different performance and security guarantees.

6 Evaluation

We evaluate RSTI by answering the following questions:

- How effective is RSTI in preventing state-of-the-art attacks as well as real-world attacks? (§6.1)
- How secure is RSTI against abusing pointer-to-pointer metadata and pointers with the same RSTI-type? (§6.2)
- How much performance overhead does RSTI impose in benchmarks and real-world programs? (§6.3)

6.1 Security Evaluation

This section evaluates RSTI's effectiveness in stopping security attacks. We first explain the different types of attacks (§6.1.1), and how RSTI defends against these attacks (§6.1.2).

6.1.1 Attacks Landscape. In this section, we show that RSTI can defend against both control-flow hijacking and data-oriented attacks. Table 1 shows a variety of state-of-the-art attacks, grouped by category, and includes both real-life software code (R) and synthetic victim code (S) attacks. Real-life software code attacks are attacks performed on actual software with actual vulnerabilities, and synthetic victim code attacks are a contrived exploit of the given class. We exercised RSTI against powerful attacks, such as NEWTON [81] and AOCR [69], as well as C++-specific attacks, such as COOP. We also chose attacks from exploits in `libtiff` and Python. Lastly, we evaluate RSTI against a Data Oriented Programming (DOP) attack [44] that leaks an SSL key. We surveyed these latest state-of-the-art attacks and made our best effort to include all the latest attacks that cover all the aspects that we want to showcase, such as use of code pointers, data pointers, and real-life code vs. synthetic victim code. The COOP attacks are synthetic victim code attacks, while the rest are executed on real-life software code. There are no standard security benchmarks for evaluating data pointers.

Table 1. Real and synthesized exploits. This table shows which pointers are being abused and how the scope-type information changes. **(R)** refers to attacks on real-life software code. **(S)** refers to attacks on synthetic victim code.

Attack Category & Type		Pointers being abused	Original scope-type information	Corrupted scope-type information
Control flow hijacking	NEWTON CsCFI attack [81] **(R)**	**Corrupted:** `c->send_chain` **Target:** `malloc`	**Type:** `ngx_send_chain_pt` **Scope:** `ngx_http_write_filter`	**Type:** `void* (size_t size)` **Scope:** `libc`
	AOCR NGINX Attack 1 [69] **(R)**	**Corrupted:** `task->handler` **Target:** `_IO_new_file_overflow`	**Type:** `void (*handler) (void *data, ngx_log_t *log)` **Scope:** `ngx_thread_pool_cycle`	**Type:** `int *(File *f, int ch)` **Scope:** `libc`
	AOCR NGINX Attack 2 [69] **(R)**	**Corrupted:** `p=log->handler` **Target:** `ngx_master_process_cycle`	**Type:** `ngx_log_writer_pt` **Scope:** `ngx_log_set_levels`	**Type:** `void *(ngx_cycle_t *cycle)` **Scope:** `main`
	AOCR Apache Attack [69] **(R)**	**Corrupted:** `eval->errfn` **Target:** `ap_get_exec_line`	**Type:** `sed_err_fn_t` **Scope:** `sed_reset_eval, eval_errf`	**Type:** `char *(apr_pool_t *p, const char *cmd, char * const *argv)` **Scope:** `set_bind_password`
	Control Jujutsu NGINX [34] **(R)**	**Corrupted:** `ctx->output_filter` **Target:** `ngx_execute_proc()`	**Type:** `ngx_output_chain_filter_pt` **Scope:** `ngx_output_chain`	**Type:** `static void *(ngx_cycle_t *cycle, void* data)` **Scope:** `ngx_execute`
	CVE-2014-8668 **(R)**	**Corrupted:** `tif->tif_encoderow` **Target:** Arbitrary pointer	**Type:** `TIFFCodeMethod` **Scope:** `_TIFFSetDefaultCompression, TIFFWriteScanline, TIFFOpen, main`	Unknown, since this is a CVE and not an attack
	CVE-2014-1912 **(R)**	**Corrupted:** `tp->tp_hash` **Target:** Arbitrary pointer	**Type:** `hashfunc` **Scope:** `inherit_slots, PyObject_Hash`	Unknown, since this is a CVE and not an attack
	COOP REC-G [27] **(S)**	**Corrupted:** `objB->unref` **Target:** `virtual ~Z()`	**Type:** `class X` **Scope:** `class Z`	**Type:** `class Z` **Scope:** `class Z`
	COOP ML-G [73] **(S)**	**Corrupted:** `students[i] ->decCourseCount()` **Target:** `virtual ~Course()`	**Type:** `void *()` **Scope:** `class Student, class Course`	**Type:** `class Course` **Scope:** `class Course`
	PittyPat COOP Attack [31] **(S)**	**Corrupted:** `member_2->registration` **Target:** `member_1->registration`	**Type:** `void*()` **Scope:** `main, class Student`	**Type:** `void*()` **Scope:** `main, class Teacher`
Data oriented attack	DOP ProFTPd Attack [44] **(R)**	**Corrupted:** `&ServerName` **Target:** `resp_buf, ssl_ctx`	**Type:** `const char*` **Scope:** `core_display_file`	**Type:** `char*` **Scope:** `pr_response_send_raw`
	NEWTON CPI Attack [81] **(R)**	**Corrupted:** `v[index].get_handler` **Target:** `dlopen`	**Type:** `ngx_http_get_variable_pt` **Scope:** `ngx_http_get_indexed_variable`	**Type:** `void* (const char *filename, int flags)` **Scope:** `ngx_load_module`

6.1.2 Attacks in Detail and How RSTI Defends.

As Table 1 indicates, in all the known attacks, the scope-type information of the corrupted pointers differs from the original one, thus allowing the attacks to be detected by RSTI. Due to space limitations, we only explain two of these attacks in detail.

NEWTON CsCFI attack. This is an attack on NGINX that calls and exploits `mprotect`. One of the attack steps is maliciously overwriting a function pointer (`c->send_chain`) in the function `ngx_http_write_filter` with a pointer to `malloc`. If the `malloc` pointer is in `libc`, then it won't have a PAC on it, and thus authentication would fail. Moreover, if we assume that both the function pointer and the `malloc` pointer are protected by RSTI, then authentication would still fail due to both having different types (`void*(size_t size)` and `ngx_send_chain_pt`), as well as being in different scopes. Thus, RSTI detects this attack.

DOP ProFTPd attack. This is a Data Oriented Programming attack [44] which corrupts the first known pointer of struct `ssl_ctx` in a loop and overwrites it with 8 malicious dereferences, relying on 4 gadgets for each dereference. Since these dereferences invoke `load` gadgets, these gadgets are protected by RSTI, and thus cannot be invoked maliciously without conforming to the RSTI-type. This load gadget corrupts `&ServerName` with data from `resp_buf`. However, `&ServerName` and `resp_buf` have different RSTI-types. `&ServerName` is of type `const char*` and its scope is `core_display_file`, while `resp_buf` is of type `char*` with scope `pr_response_send_raw`. Since RSTI protects data pointers with RSTI-type, RSTI detects this attack. This attack can only succeed if all dereferences and gadgets have the same RSTI-type.

Table 2 summarizes the RSTI mechanisms, attacker restrictions, and defense capabilities of each mechanism.

Comparison against prior works. RSTI offers a refined type compared to other prior data pointer integrity works that use PAC, such as PARTS [55], and thus has better coverage against pointer substitution attacks. For example, in the DOP ProFTPD attack in Table 1, the corrupted and original pointers are both of type `char*`. PARTS wouldn't be able to detect this, since it only relies on type. However, RSTI's refined scope-type detects it. The PittyPat attack is another example. Thus, RSTI can mitigate more attacks.

6.2 Analysis on RSTI Instrumentation

6.2.1 Equivalence Class.

Here, we showcase the usefulness of RSTI-type information. We define some terminology:

Table 2. Security evaluation summary. This table shows how each RSTI mechanism constricts an attacker, as well as their security guarantees.

Technique	RSTI-STC	RSTI-STWC	RSTI-STL
How it works	Constricting variables to their scope, type and permission, and handle casts by combining into one compatible type	Constricting variables to their scope, type and permission, and handle casts by authenticating/re-signing a pointer	Constricting variables to their scope, type, permission and location without combining. In other words, adding &p to the modifier.
Attacker restriction	The attacker here can substitute pointers if they both have a valid PAC and as well as have the same RSTI-type.	Similar to RSTI-STC, the attacker can substitute pointers if they both have a valid PAC, as well as have the same RSTI-type.	The attacker here cannot even substitute a pointer at a separate location.
Defense capability	**Pointer corruption:** An attacker cannot substitute with pointers that have different RSTI-types. However, the size of the RSTI-type may be large due to combining. **Spatial Safety:** An attacker overflowing the buffer needs to override with a pointer or location that is of the same RSTI-type. **Temporal Safety:** Similar to spatial safety, an attacker needs to reuse the freed pointer within the same scope and type.	**Pointer corruption:** An attacker cannot substitute with pointers that have different RSTI-types. Due to not combining, RSTI-STWC is stronger than RSTI-STC. **Spatial Safety:** Achieving an attack through spatial violations is harder, due to stricter RSTI-type at that pointer. **Temporal Safety:** Similarly, an attacker needs to reuse the freed pointer with the same RSTI-type, but stronger than RSTI-STC.	**Pointer corruption:** Here, the pointer would have no substitutes, meaning that only that pointer can be dereferenced from that location. **Spatial Safety:** It wouldn't be possible to abuse a pointer with a spatial safety violation. A buffer overflow would always be detected due to &p. **Temporal Safety:** Similar argument to spatial safety, any usage of the freed pointer with any other pointer or at another location would be detected due to &p.

Table 3. SPEC 2006 equivalence class data. (NT: Number of types in the program; RT: Number of RSTI-types; NV: Total number of pointer variables; EC_V: Equivalence class of variable; EC_T: Equivalence class of type.)

BM	NT	RT		NV	Largest EC_V		Largest EC_T	
		STC	STWC		STC	STWC	STC	STWC
perlbench	155	318	722	2939	198	82	33	1
bzip2	25	31	55	122	32	13	7	1
mcf	12	35	40	95	9	8	2	1
milc	55	154	195	440	54	18	18	1
namd	30	73	100	230	23	23	10	1
gobmk	120	216	417	1057	111	46	25	1
dealII	2546	4528	8878	21018	676	44	192	1
soplex	129	970	1690	3399	137	27	66	1
povray	282	620	1446	3791	229	25	76	1
hmmer	90	198	405	973	56	24	16	1
libquantum	13	33	44	58	9	4	5	1
sjeng	29	47	73	130	19	9	7	1
h264ref	116	252	354	727	48	23	15	1
lbm	14	14	20	33	12	7	4	1
omnetpp	255	558	1241	2458	94	26	31	1
astar	36	59	98	156	18	11	12	1
sphinx3	88	188	321	686	36	20	12	1
xalancbmk	2558	7503	14073	32097	603	122	206	1

- **Number of types in program (NT)**: This is the number of basic types a program has, such as `int*`, `void*`, etc.
- **Number of RSTI-types (RT)**: This refers to the actual types that are enforced by the specific RSTI mechanism.
- **Equivalence Class of Type (EC_T)**: RSTI-STWC has the advantage of having only *one basic type* for each *RSTI-type*, whereas RSTI-STC can have more than one basic type in an RSTI-type, due to the combining. We refer to the number of basic types in each RSTI-type as *Equivalence Class of Type (EC_T)*. We use the term *equivalence class* since this count provides a measure of how viable pointer substitution attacks can be within an application.
- **Equivalence Class of Variable (EC_V)**: RSTI-STWC has a maximum EC_T of 1, due to there being only one type in each RSTI-type. However, RSTI-STWC does not have *one variable* for each RSTI-type. Several variables can be declared within the same RSTI-type. We refer to the number of variables in each RSTI-type as *Equivalence Class of Variable (EC_V)*. The largest EC_V for RSTI-STL is always 1, due to the inclusion of the location (`&p`) in the modifier, while the largest EC_V for RSTI-STWC would vary.

Table 3 shows the EC data for SPEC CPU2006. RSTI increases the number of types that can be distinguished in a program. Also, the impact of combining RSTI-types for RSTI-STC reduces the number of RSTI-types in the mechanism, but is still higher than the number of types without RSTI. RSTI-STL is not shown in the table. This is because the largest EC_T and EC_V for RSTI-STL are always 1, due to the enforcement of the location. RSTI-STL is the most secure, however RSTI-STWC does still provide security value due to its EC_T being always 1, and significantly reducing EC_V. In regards to the PAC length, prior work [42] has shown that the number of PACs available is sufficient for practical cases.

6.2.2 Pointer-to-pointer data from SPEC 2006.

We had previously explained our exact use case of pointer-to-pointer handling in §4.7.7. Our intuition was that this exact case of losing the original type of the pointer when a pointer-to-pointer is casted and passed as an argument to a function is rare. This was confirmed by our analysis of SPEC 2006. There is a total of 7,489 sites across the benchmarks where a pointer-to-pointer is passed or loaded. Of those, only 25 meet the special criteria where the original type could be lost. This confirms our intuition that this is a rare case.

6.3 Performance Evaluation

6.3.1 Methodology.

Our evaluations were run on the Apple M1 [9], which supports the ARMv8.4 architecture and ARM PA. We used an Apple Mac Mini M1 [8], that has 4 small cores, 4 big cores, and 8GB DRAM. Our prototype was implemented on Apple's LLVM fork [1]. We compiled all the applications with `LTO` and `O2` for fair comparison.

Evaluating C programs. All our C programs were run with real PA instructions. We were able to disable Apple's use of PA [10] to avoid any conflicts that would happen between RSTI's and Apple's PA instrumentation.

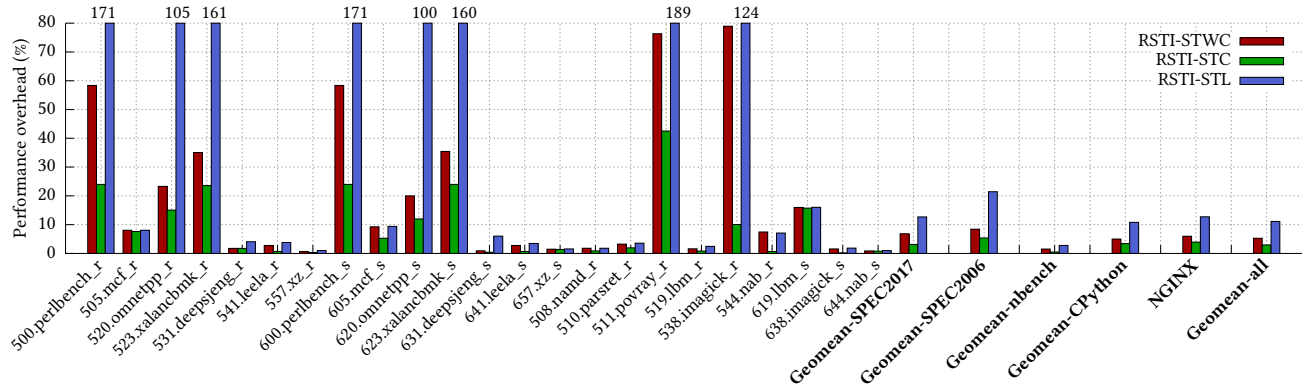


Figure 9. The performance overhead of SPEC CPU2017, and the geometric means of SPEC CPU2006, nbench, CPython Pytorch and NGINX for all three RSTI mechanisms.

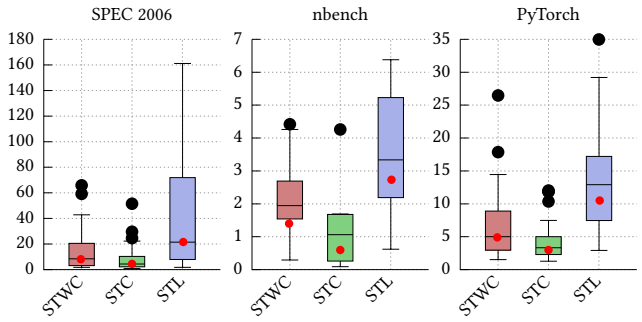


Figure 10. The performance overhead and geometric means of SPEC CPU2006, nbench and CPython Pytorch for all three RSTI mechanisms. The box plot shows the minimum, median, maximum and quartile values. The black dots represent outlier values. The red dots represent the geometric mean.

Evaluating C++ programs. Whilst evaluating C++ programs, we discovered that PA instructions are built into Apple's standard C++ library. Thus, we were not able to turn them off. So, for C++ applications, we first did a correctness test on ARM's Fixed Virtual Platform (FVP) [11]. Then, in order to emulate the performance overhead of the PA instructions, we used seven XOR (`eor`) instructions as an equivalent to one PA instruction on the Mac Mini M1. This has been measured and confirmed in previous works [47, 54, 84].

Benchmarks. For our performance evaluation, we used a wide variety of benchmarks to showcase RSTI's versatility, namely: SPEC CPU 2006 [41], SPEC CPU 2017 [16], and nbench [59]. We ported the SPEC CPU 2006 benchmarks to the Apple M1 and built them from scratch. We were not able to run `403.gcc` and `625.x264` on the M1, in spite of using Apple's own Clang compiler. We think that there is a bug in the macOS toolchain version that we used.

6.3.2 Performance Overhead.

Figure 9 shows the performance overhead for the SPEC CPU2017 benchmarks, as well as NGINX and the geometric mean of SPEC CPU2006, nbench and CPython PyTorch. The individual performance numbers for SPEC CPU2006, nbench and CPython PyTorch were not included due to space limitations. Thus, we use boxplots to show the distribution in Figure 10.

SPEC CPU2017. SPEC 2017 benchmarks have a geometric mean of 6.86%, 3.17%, and 12.70% for RSTI-STWC, RSTI-STC, and RSTI-STL respectively. Some benchmarks, such as `perlbench`, `povray`, and `xalancbmk` have exceptionally higher overhead. These benchmarks are known to heavily dereference pointers, either in a loop or very frequently [47].

SPEC CPU2006. SPEC 2006 benchmarks have a geometric mean of 8.42%, 5.36%, and 21.47% for RSTI-STWC, RSTI-STC, and RSTI-STL, respectively. We analyzed the instrumentation and found the performance overhead is highly correlated with the number of instrumented load/stores, with a Pearson correlation factor of 0.75-0.8. There are some exceptions, due to loops or some of the instrumented loads/stores never getting called, but the overall results are consistent.

CPython 3.9. For CPython3.9, we evaluated PyTorch benchmarks [3] to test how RSTI would perform if used in a machine learning context.The CPython PyTorch benchmarks have a geometric mean of 5.01%, 3.44%, and 10.80% for RSTI-STWC, RSTI-STC, and RSTI-STL, respectively. Thus, RSTI can be utilized in machine learning scenarios where additional security guarantees are needed.

NGINX. We evaluated NGINX, a real-world application, on the Apple M1. We stress its 4 big cores and use the same configuration used to test NGINX TLS transactions per second [60]. We relied on `wrk` [36], an HTTP benchmarking tool, to stress test NGINX by generating concurrent HTTP requests. `wrk` was run on a separate machine on the same network and spawned three threads, with each thread handling 50 connections. The overhead was 5.98%, 3.93%, and 12.76% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

The total geometric mean across all the benchmarks and applications is 5.29%, 2.97% and 11.12% for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

Overall. As can be seen, the RSTI mechanisms have mild overhead. Some exceptions are there for RSTI-STL, where the overhead can exceed 100%. This is still comparatively lower to other mechanisms such as DFI, which has an average overhead of around 106%. However, real programs, such as NGINX and PyTorch, have a reasonable overhead. Thus, the RSTI mechanisms can be effectively used to secure machine learning and real-world applications.

Comparison against prior works RSTI offers lower overhead compared to other PAC-based defense mechanisms such as PARTS [55]. PARTS is evaluated only on nbench, and the mean overhead is 19.5%. In comparison, RSTI's mean overhead on nbench is 1.54%, 0.52% and 2.78% for RSTI-STWC, RSTI-STC and RSTI-STL respectively. Although both PARTS and RSTI instrument data pointers, our design choices, such as using LLVM `ptrauth` intrinsics, running the pass in the backend, using LTO and `-O2` optimizations allowed our compiler to produce more optimized code.

7 Discussion and Limitations

Comparison with memory safety. Due to the constricting nature of the RSTI mechanisms and due to the fact that RSTI protects all pointers in a program, RSTI provides a static alternative to traditional memory safety techniques for providing spatial and temporal safety. Bear in mind that RSTI does not prevent spatial and temporal memory errors but prevents an attacker from abusing them. In addition to that, RSTI does not instrument non-pointer variables, and they are out of scope of this work. The design can be extended to include offsets and indexes by converting them to pointers to cover them, but we relegate this to future work.

Metadata attack in RSTI. Pointer-to-pointer metadata is the only metadata that is stored in memory. However, the attacker cannot see the actual types in memory. In our implementation, each type is assigned a type ID in the internal LLVM data structure during compilation. When storing pointer-to-pointer metadata, that type ID is used to represent each type. Thus, the metadata information is not meaningful to the attacker. Also, the metadata is read-only and can only be accessed by the RSTI pointer-to-pointer library.

RSTI with mechanisms other than PAC RSTI can be enforced without necessarily relying on ARM's PAC. PAC is more of an implementation choice. The enforcement can be done with any mechanism that can utilize the scope-type information. For example, CCFI [58] relies on classes of pointers and an AES cryptographic function to generate MACs that get stored alongside the object. A hardware-accelerated AES cryptographic mechanism, similar to the one in CCFI, can be used to replace PAC. RSTI can also be used in embedded systems by using PACBTI [13] introduced in ARMv8.1-M. It is similar to ARMv8.3 PAC, but the PAC is placed in a separate register and not on the pointer.

Possibility of replay attacks Due to the existence of an equivalence class more than 1 for RSTI-STC and RSTI-STWC, as can be seen in Table 3, it would be theoretically possible for replay attacks to occur. For example, an attacker wanting to abuse `perlbench` under RSTI-STWC would have to choose gadgets that are confined to the 82 equivalent variables. This is even more of a possibility with RSTI-STC due to the combining of RSTI-types. For example, an attacker wanting to abuse `perlbench` under RSTI-STC would have to choose gadgets that are confined to the 33 equivalent types or the 198 equivalent variables. However, the viability of a replay attack is still reduced with RSTI. In order to carry out a successful attack by abusing pointers in a program, an attacker needs the pointers to be actually useful to the attack, not just any pointers. This makes practical attacks harder to execute on the RSTI-mechanisms. A potential area of future work might be choosing the mechanism based on the variables with the same RSTI-type. For example in the case of `xalancbmk` which has 122 equivalent variables for RSTI-STWC, STL can be used to ensure security. RSTI-STWC can be used when the number of variables with the same RSTI-type is smaller, such as in the case of `mcf` which has only 9 equivalent variables for RSTI-STWC.

Handling external code RSTI supports uninstrumented libraries by stripping the PAC when an external library call is made. However, this does not work for all cases. If a pointer is passed directly to the external library, then the pointer will be authenticated first, and the same happens when the pointer gets passed from the external library. However, if there is a composite pointer, for example a pointer to a structure that is a node in a linked list, then this would not be supported without compiling the external library with RSTI to verify the pointer when it gets used in the external library. If the external library is not compiled with RSTI, then it is a bad pointer that will cause the RSTI security check to fail. Thus, the external library could be compiled with RSTI if needed. This issue is also available in other similar solutions such as CPI [53] and VIP [48].

Exclusion of non-pointer variables RSTI only instruments pointer variables. Non-pointer variables are out of scope. The design can be extended to include converting offsets to pointers to cover them. However, this is not in the scope of this work and we relegate this to future work.

8 Related Work

Type-based defenses. EffectiveSan [32] conducts bounds checking by combining type checking with low fat pointers. RSTI does not aim to do bounds checking and does not rely on low fat pointers. TDI [62] relies on grouping types into specific memory arenas by relying on a special allocator and compiler instrumentation. RSTI doesn't need a special allocator and thus doesn't suffer from the same compatibility issues. Type after type [79] replaces regular allocations with

typed allocations that never reuse memory previously used. Since this is done with a special allocator, it has compatibility issues that RSTI doesn't suffer from. Type-based defenses suffer from attacks that abuse pointers of the same type. RSTI adds the permission, scope and location to better defend against these attacks. However, this means that external libraries must be instrumented with RSTI to be protected. TDI, for example, doesn't require that since pointers passed to external functions or stored in memory are always masked. This limitation can be overcome by compiling the library with the RSTI compiler.

ARM PAC defenses. PARTS [55] protects pointers with pointer authentication by relying on the LLVM `ElementType` as the modifier. The `LLVMElementType` is likely to repeat and can be exploited [47]. PARTS relies only on the default type and doesn't have the distinct type of scope-type information that RSTI provides. This makes PARTS vulnerable to pointer substitution attacks. RSTI is less susceptible to these attacks than PARTS. PARTS has an average overhead of 19.5% on nbench, while RSTI has an averageof 1.54%, 0.52%, and 2.78% on nbench for RSTI-STWC, RSTI-STC, and RSTI-STL respectively. AOS [51] utilizes the PAC as an index to access a bounds table and do bounds checking. AOS extends the ISA with extra instructions and only handles the heap, whereas RSTI doesn't need any custom extensions and handles both heap and stack. AOS does ensure heap spatial and temporal safety, whereas RSTI does not. However, this comes at the cost of modifying the hardware, whereas RSTI does not. This tradeoff makes RSTI more practical to implement.

Data oriented defenses. Data Flow Integrity (DFI) [22] makes sure that a variable can only be written by its legitimate write instruction. However, complete DFI incurs around 103% runtime overhead on average and 50% memory overhead. By comparison, RSTI has a lower average runtime overhead, and almost no memory overhead. Hardware assisted Data Flow Isolation (HDFI) [75] provides instruction-level isolation by relying on tags. Whenever a memory is read, HDFI checks if the tag matches the expected value. However, HDFI relies on a 1-bit tag, and thus only two protection domains. RSTI's use of pointer authentication allows for a much finer grained protection. YARRA [72] is an extension of the C language that relies on programmer annotations to protect critical data types. By comparison, RSTI does not need any programmer annotations. WIT [7] uses points-to analysis to identify objects that can be modified by each instruction. They then use static analysis to identify memory accesses and objects that are safe. These are accesses that do not violate write integrity and objects that only have safe accesses. WIT only instruments writes that are unsafe. However, WIT does not protect reads, and thus data can be corrupted when being read into a register. RSTI can defend against this type of attack. RSTI protects all reads and writes in a program and makes sure they are legitimate.

Control-flow hijacking defenses. Control-Flow Integrity (CFI) [6] constricts the number of valid target sites for an indirect control-flow transfer. Static CFI schemes are vulnerable to a control flow bending [19] attack. Since RSTI protects all pointers, RSTI is able to defend against a bending attack, whereas static CFI schemes cannot. Code Pointer Integrity (CPI) [53] protects a subset of pointers, referred to as sensitive pointers. Since RSTI protects all pointers, it provides stronger guarantees than CPI. CPI is vulnerable to the NEWTON attack, whereas RSTI is able to defend against it as we discussed in §6.1.

Other memory safety defense mechanisms Both RSTI and memory safety techniques aim to defend against attackers by enforcing the correct execution of a program. However, RSTI does not prevent spatial and temporal memory errors and it does not protect non-pointer variables, compared to memory safety techniques that might cover these. The trade-off here is that RSTI has much more efficient performance than these techniques whilst providing protection to all pointers in a program. Our choice of context is built off of what is already available in the program, so it does not need complex compiler analysis to determine how the pointers should be used.

It is possible for other memory safety guarantees to be encoded in PAC contexts. For example, PACMem [54] uses the object size and a unique birthmark to encode the metadata for memory objects into the PAC. However, this necessitates a large and complex metadata with memory overhead of around 82.96%, whilst RSTI uses little metadata. SAFECode [30] is a heap-based memory safety mechanism that uses pool allocation based on types. RSTI is able to encode the metadata it needs into the pointer, and thus can do runtime checks directly with the authentication instructions, whereas SAFECode relies on doing heap runtime checks due to its reliance on the pool allocation. In addition to that, RSTI's implementation is simpler than SAFECode. There could be some errors that are detected by RSTI which are not detected by SAFECode, and vice versa. However, we haven't been able to measure this. Baggy Bounds Checking (BBC) [4] is a bounds checking mechanism that relies on checking allocation bounds rather than precise object bounds. PAMD [57] extends BBC to attach metadata to memory objects. SoftBound+CETS [63] rely on bounds checking and a lock-and-key mechanism to ensure temporal and spatial safety. In contrast to these mechanisms, RSTI doesn't do bounds checking and focuses on protection of pointers. Even though RSTI does not guarantee spatial and temporal safety to the extent that these mechanisms do, it still does provide some spatial and temporal safety guarantees. SAFECode, BBC and PAMD do not require external libraries to be recompiled, where as RSTI does in order to fully protect them due to the change in pointer layout. Due to the change in pointer semantics, it is necessary for external libraries to

have the PAC instructions instrumented in order to protect them. However, the tradeoff here is that this comes at the cost of non-negligible memory overhead due to the extra allocations in memory needed for the runtime checks. SoftBound+CETS doesn't change the pointer layout but bounds information still is not propagated to the external libraries. Thus, the limitation of external library compatibility is not unique to RSTI. CHERI [83] redefines pointers into capabilities, providing security with additional embedded metadata. CHERI completely redefines what a pointer is, thus having compatibility issues that RSTI doesn't have. HardBound [29] is a memory safety defense mechanism that does hardware bounds checking. HardBound requires new hardware instructions to be added, while RSTI can defend with already existing hardware security features. BOGO [87] utilizes Intel MPX [46] to achieve temporal safety, in addition to the spatial safety provided by MPX. BOGO incurs a mean of 60% runtime overhead for SPEC CPU 2006 benchmarks, whilst RSTI incurs a mean of 8.42%, 5.36%, and 21.47% for RSTI-STWC, RSTI-STC, and RSTI-STL, respectively.

9 Conclusion

This paper introduced Scope-Type Integrity (STI), a new defense policy that enforces pointers to conform to the programmer's intended usage by utilizing scope, type and permission information and bringing that information back to leverage during runtime. We presented RSTI, a robust and efficient security mechanism that protects all pointers in a program by leveraging ARM Pointer Authentication. RSTI enforces STI to ensure that each pointer conforms to its scope, type, and permission that was originally intended by the programmer. RSTI leverages this bought back information in the runtime. We implemented three RSTI defense mechanisms with varying levels of security granularity, enforcing control-flow and data-flow integrity. We showcased the security of RSTI against state-of-the-art synthesized and real attacks, and demonstrated its low performance overhead across a variety of benchmarks and real-world applications.

Acknowledgments

We thank our shepherd, John Criswell, and the anonymous reviewers for their insightful comments and input to improve the paper. We also thank the artifact evaluators and Jack Chandler for their efforts. This work was supported by the National Science Foundation (NSF) under grant 2153748.

A Artifact.

A.1 Abstract

This artifact contains the source code for the RSTI compiler, the CMake file needed to compile it, and example C and C++ programs to test it. It should be noted that this source code needs an Apple M1 machine with MacOS to properly test it. The RSTI compiler is built on Apple's fork of the LLVM compiler infrastructure and uses hardware security features that are available on the M1 chip. The artifact contains scripts to compile the RSTI compiler, as well as scripts to compile the example programs.

A.2 Artifact check-list (meta-information)

- **Compilation:** Compiling the RSTI Clang compiler and C/C++ code examples compiled the RSTI compiler.
- **Run-time environment:** MacOS 12 - Monterey.
- **Hardware:** Mac Mini with Apple M1 chip, 8GB RAM, 512GB SSD.
- **Execution:** Running compiled sample programs.
- **Output:** Generated binaries of the sample programs in the same directory.
- **How much time is needed to prepare workflow (approximately)?:** Around 20-30 minutes is needed to compile the custom LLVM.
- **Publicly available?:** Yes, it is publicly available at: https://doi.org/10.5281/zenodo.10799158. The github repo is also available at https://github.com/cosmoss-jigu/rsti
- **Code licenses (if publicly available)?:** Apache License, Version 2.0
- **Archived:** 10.5281/zenodo.10799158

A.3 Description

A.3.1 How to access. The artifact can be accessed from the following link: https://doi.org/10.5281/zenodo.10799158. It can also be accessed through this Github repo by cloning it: https://github.com/cosmoss-jigu/rsti.

A.3.2 Hardware dependencies. The artifact requires the Apple M1 chip with the ARMv8.3 ISA or later.

A.3.3 Software dependencies. The artifact requires MacOS with cmake, make and git packages installed. In addition, the artifact requires System Integrity Protection (SIP) to be disabled, and to add support for `arm64e` compilation to the `nvram` boot arguments. Please check the README.md file in the Github repo for instructions on how to do this.

A.4 Installation

Extract the tar ball from Zenodo or clone the Github repo to your machine. Then follow the instructions listed in the README.md file.

A.5 Experiment workflow

1. Create a build folder in the directory with the command `mkdir build`
2. Go into the build folder with the command `cd build`
3. Generate the Makefile from the CMakeLists with the command `cmake ..`
4. Compile the RSTI compiler with the command `make llvm-mac-all`
5. Run the `rsti_compile` scripts with their respective example C/C++ program.
6. Run the generated binaries of each example program.

For more details, see the README.md file.

A.6 Evaluation and expected results

Success in running the compiled programs with a "Hello World" message being printed for the C program, and a "Hello from regular function" message being printed for the C++ program

A.7 Notes

If you wish to use the RSTI compiler, please check the compile scripts in the example directory for all the flags necessary. You can copy the file and modify it as needed.

References

[1] Apple LLVM. https://github.com/apple/swift-llvm.

[2] Llvm pointer authentication. https://llvm.org/docs/PointerAuth.html.

[3] PyTorch Benchmarks. https://github.com/pytorch/benchmark.

[4] Baggy bounds checking: An efficient and Backwards-Compatible defense against Out-of-Bounds errors. In *18th USENIX Security Symposium (USENIX Security 09)*, Montreal, Quebec, August 2009. USENIX Association.

[5] Clang 13. ShadowCallStack, 2021. https://clang.llvm.org/docs/ShadowCallStack.html.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277, 2008.

[8] Apple. Apple Mac Mini M1, 2020. https://www.apple.com/shop/buy-mac/mac-mini/apple-m1-chip-with-8-core-cpu-and-8-core-gpu-256gb.

[9] Apple. Apple unleashes M1, 2020. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[10] Apple. Operating system integrity, 2021. https://support.apple.com/en-hk/guide/security/sec8b776536b/1/web.

[11] Arm. Fixed Virtual Platforms. https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms.

[12] Arm. Top Byte Ignore, 2018. https://en.wikichip.org/wiki/arm/tbi.

[13] Arm. PACBTI, 2020. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension.

[14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[15] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, page 30–40, Hong Kong, China, March 2011.

[16] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[18] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[20] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 193–204, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, Seattle, WA, November 2006.

[23] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[24] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[25] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[26] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[27] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection Against Function-reuse Attacks. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[28] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[29] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 103–114, 2008.

[30] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 144–157, New York, NY, USA, 2006. Association for Computing Machinery.

[31] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 131–148, Vancouver, BC, Canada, August 2017.

[32] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson,

Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, page 901–913, Denver, Colorado, October 2015.

[35] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[36] Will Glozer. a HTTP benchmarking tool, 2019. https://github.com/wg/wrk.

[37] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[38] Jens Grossklags and Claudia Eckert. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Heraklion, Crete, Greece, September 2018.

[39] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[40] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery.

[41] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[42] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Pacsafe: Leveraging arm pointer authentication for memory safety in c/c++, 2022.

[43] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[44] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[45] Qualcomm Technologies Inc. Pointer Authentication on ARMv8.3, 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[46] Intel. Support for Intel® Memory Protection Extensions (Intel® MPX) Technology, 2015. https://www.intel.com/content/www/us/en/support/articles/000059823/processors.html.

[47] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with PACTight. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3717–3734, Boston, MA, August 2022. USENIX Association.

[48] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Vip: Safeguard value invariant property for thwarting critical memory corruption attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, 2021.

[49] Jonathan Corbet. x86 NX support, 2004. https://lwn.net/Articles/87814/.

[50] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[51] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1153–1166, 2020.

[52] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[53] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[54] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 177–194, Santa Clara, CA, August 2019.

[56] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[57] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. page 36–46, jun 2017.

[58] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[59] Uwe Mayer. Linux/Unix nbench, 2017. https://www.math.utah.edu/~mayer/linux/bmark.html.

[60] Faisal Memon. NGINX Plus Sizing Guide: How We Tested, 2016. https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/.

[61] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[62] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1049–1065, 2022.

[63] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[64] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[65] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[66] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In

Proceedings of the 22th USENIX Security Symposium (Security), Washington, DC, August 2013.

[67] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[68] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 685–698, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[70] Jonathan Salwan. Ropgadget: Gadgets finder and auto-roper, 2019. https://github.com/JonathanSalwan/ROPgadget.

[71] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 131–145, 2011.

[72] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[73] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[74] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[75] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Pack. HDFI: Hardware-Assisted Data-flow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[76] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[77] The Clang Team. Clang 10 documentation: CONTROL FLOW INTEGRITY, 2019. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[78] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[79] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 17–27, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[81] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[82] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[83] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News*, page 457–468, jun 2014.

[84] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, Boston, MA, August 2022. USENIX Association.

[85] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[86] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[87] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.