HOBBIT: Hashed OBject Based InTegrity

Matthias Bernad
μCSRL – Munich Computer Systems Research Lab, Research Institute CODE
University of the Bundeswehr Munich, Neubiberg, Germany

Stefan Brunthaler
μCSRL – Munich Computer Systems Research Lab, Research Institute CODE
University of the Bundeswehr Munich, Neubiberg, Germany

Abstract

C vulnerabilities usually hold verbatim for C++ programs. The *counterfeit-object-oriented programming* attack demonstrated that this relation is asymmetric, i.e., it only applies to C++. The problem pinpointed by this COOP attack is that C++ does not validate the integrity of its objects. By injecting malicious objects with manipulated virtual function table pointers, attackers can hijack control-flow of programs. The software security community addressed the COOP-problem in the years following its discovery, but together with the emergence of transient-execution attacks, such as Spectre, researchers also shifted their attention.

We present HOBBIT, a software-only solution to prevent COOP attacks by validating object integrity for virtual function pointer tables. HOBBIT does not require any hardware specific features, scales to multi-million lines of C++ source code, and our LLVM-based implementation offers a configurable performance impact between 121.63% and 2.80% on compute-intensive SPEC CPU C++ benchmarks. HOBBIT's security analysis indicates strong resistance to brute forcing attacks and demonstrates additional benefits of using execute-only memory.

2012 ACM Subject Classification Security and privacy → Software security engineering; Software and its engineering → Compilers

Keywords and phrases software security, code-reuse attacks, language-based security, counterfeit-object-oriented programming, object integrity, compiler security.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.41

Funding The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) [(FFG grant no. 888338)] within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

1 Motivation

Among the myriad of security exploits, control-flow hijacking is the most severe problem, as it allows the attacker to execute arbitrary code. A buffer overflow, for example, allows an attacker to overwrite the return address stored in a function's stack frame, and thus divert control-flow to a location of her choice. Many other similar vulnerabilities exist and have been both explored and exploited over the past two decades. Most of these vulnerabilities affect both C and C++ alike.

The feasibility of an attack focusing exclusively on the C++ superset was demonstrated by Schuster et al. in 2015 [43]. By injecting malicious objects into a C++ application the attack hijacks control-flow and allows Turing-complete, arbitrary computation. In analogy to other similar attacks, such as return-oriented programming, this attack is known as *counterfeit-object-oriented programming*, COOP for short.

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 41; pp. 41:1–41:25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Due to the prevalence of C++ in systems and application software, researchers focused on devising mitigations against COOP. To prevent control-flow hijacking, prior defenses apply principles from effective C defenses. The principle of W^X limits attacker capabilities to inject code through new hardware features, such as Intel's NX bits [50]. CFIXX, for example, uses the MPX extension to secure a bookkeeping table it relies upon [10]. By applying cryptography to encode and decode control-flow data, adversaries cannot know a priori how target addresses are encoded. CCFI, for example, uses Intel's AES-NI instructions to cryptographically secure program addresses, such as return addresses, function-, and `vtable` pointers [29].

Although both of these defenses thwart COOP attacks, they, too, have drawbacks. Reliance on Intel's MPX is problematic for three reasons. First, MPX may not be available in a system's target environment, such as in embedded systems or IoT contexts. Second, Intel could decide to abandon the MPX instruction set extensions. Consider the MPK instruction set extension, which was discontinued rather abruptly, rendering defenses relying on it incapacitated. Third, MPX is non-compositional: A defense cannot protect an application that already relies on MPX for its business logic, as the MPX registers are already taken.

Cryptographic protection of pointers is desirable due to strong security guarantees, but suffers from prohibitive performance penalties. CCFI's use of AES-NI reserves x86-64's vector registers, i.e., SSE, AVX, AVX2, or AVX512, blocking their use for other purposes. Unavailability affects video processing, cryptographic operations, and a variety of other tasks.

HOBBIT neither requires specific hardware extensions nor blocks vector registers and, thus, addresses both of these challenges. Instead, HOBBIT modifies the C++ object layout to embed an integrity signature when an object is constructed. This signature is validated *before* executing each virtual method's body.

A Clang/LLVM-based implementation of HOBBIT compiles large programs, such as the WebKit browser, and allows parameterization to balance security with performance. The key factor affecting performance is the choice of hashing technique to create an object's signature. Our evaluation shows that choosing strong hashing techniques can lead to substantial overheads. To eliminate this overhead, HOBBIT implements two different optimizations. First, HOBBIT applies a class-sensitive optimization to restrict its protection to classes that are essential to the COOP attack. Second, HOBBIT applies the idea of MAC algorithm parameter randomization, thereby increasing overall security. For many application contexts, HOBBIT is thus the only viable defense against COOP.

Our contributions are as follows:

- We present HOBBIT, a software-only defense that thwarts counterfeit-object-oriented programming (COOP, for short).
- We describe the implementation of a fully-fledged Clang/LLVM-based prototype that supports all C++ features, such as multiple inheritance (see Sections 5 and 6).
- We discuss two new HOBBIT optimization techniques that enable users to balance their security needs with the available performance budget. We introduce Gadget-directed optimization (see Sections 5.5 and 7.5), to apply protections specifically to COOP gadgets, and Class-Hierarchy-driven Seed Randomization (see Sections 5.3 and 6.4).
- We evaluate HOBBIT w.r.t. performance, scalability, and security (see Section 7). Specifically, we report:
  - *Performance*: A configurable performance impact between 121.63% and 2.80%.
  - *Scalability*: HOBBIT compiles complex real-world software, such as the WebKit web browser.

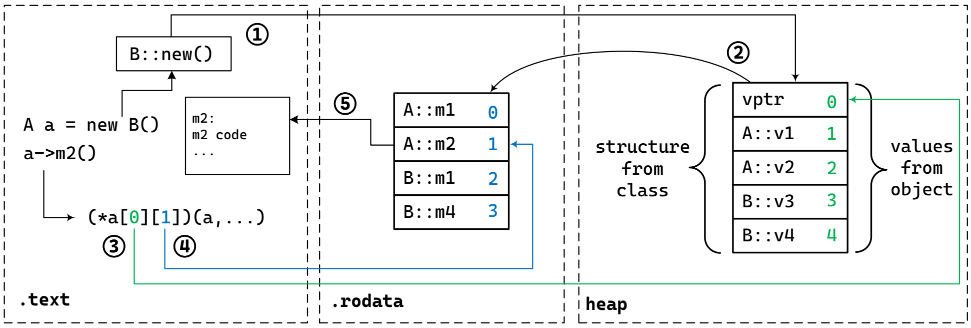


Figure 1 Overview of polymorphism and dynamic binding in C++. ① constructors allocate objects and set `vptr` and field values, ②. Method calls require resolving of the `vtable`, ③, and then the corresponding fixed method id, ④, before being able to call the method, ⑤.

- *Security*: Hobbit provides comprehensive security through either strong hashing techniques or randomizing parameters of weaker hashing techniques.

2 Background

In this section, we will introduce the background needed to understand the Counterfeit Object-Oriented Programming (COOP) attack. Since COOP is a high-level attack targeting specific C++ semantics, we will briefly explain the C++ object layout, polymorphism, and dynamic dispatch mechanism. Finally, we need to cover some preliminary concepts used in Hobbit.

2.1 C++ Polymorphism and Dynamic Binding

In object-oriented programming languages, such as C++, programs are organized around classes and objects. *Classes* in C++ define the fields of objects and the *methods* operating upon them. A concrete instance of a class is called *object* and consists of values for the defined data fields residing in a contiguous memory region. To create and initialize newly created objects, programmers call special methods, so-called *constructors*.

Figure 1 illustrates these concepts. Instantiating a new `B` object triggers a call to its constructor ①, which allocates a contiguous memory region and sets the `vptr` due to the concrete dynamic type ②. The class determines the structure of each object, while the object holds values specific to the instance.

To dynamically bind a method call, C++ uses so-called `vtable`-based method dispatch (see Figure 1). For each class, C++ generates a corresponding `vtable` that holds the addresses of each callable method on it. If a class inherits a method, its address will merely be copied into the corresponding method slot. If a class overrides a method, a new address will be written into the corresponding method slot. A method call, then, consist of two steps: (i) resolving the `vtable` by dereferencing an object an accessing the first entry, which holds the `vtable` reference ③, and (ii) resolving the method by dereferencing the proper method through a callsite-fixed method identifier ④.

2.2 Counterfeit-Object-Oriented Programming (COOP)

Over the past four decades, the memory unsafe nature of C and C++ lead to an “Eternal War in Memory” [49]. In the beginning, attackers were able to insert instructions as data in writable memory. By facilitating a buffer overflow to overwrite the return address, attackers could hijack the control-flow of a program to execute injected code, resulting in Arbitrary Code Execution (ACE). Simple defenses, such as Write exclusive-or Execute (WˆX)—marking memory as either writable or executable, but not both—render such code injection attacks impossible. Therefore, attackers adapted and began reusing existing code, residing in executable memory. Attackers either reused whole functions (e.g., return-into-libc [19, 33]) or performed arbitrary computations by chaining together small pieces of code, called *gadgets*, as in Return-Oriented Programming (ROP) [44, 40, 47] and its variations [8, 17, 13, 42]. Many defenses targeting mentioned Code Reuse Attacks (CRAs) exist [26, 35, 1, 2, 9]. A more recent CRA targeting high-level C++ semantics is COOP [43].

COOP exploits the dynamic dispatch mechanism and escapes mentioned defenses above. Instead of introducing new invalid control-flows like in ROP or return-into-libc, COOP misuses existing callsites. To illustrate this point, consider the example from the previous section, but from the perspective of the CPU. A callsite merely fixes the method identifier, but accepts *any* `vtable` and will, thus, invoke *any* method identified by the fixed method identifier (see Figure 1, ⑤.)

COOP abuses this property of `vtable`-based method dispatch, by injecting malicious objects, so-called *counterfeit objects*. These objects use invalid `vtable` entries, to abuse method invocation. Instead of abusing gadgets as in return-oriented programming, COOP abuses whole functions. Since the notion of code-reuse attacks is tied to the nomenclature of *gadgets*, COOP, too, defines whole-function reuse gadgets.

These COOP gadgets are methods that can be abused for a specific malicious purpose. Not all COOP gadgets are equally important, though. The most important gadget is the so-called *main-loop gadget*, or ML-G for short. Consider the following C++ method:

```cpp
virtual void removeElement(Element x) {
  for (int i= 0; i < this.N; i++) {
    this.L[i].remove(x);
  }
}
```

Listing 1 Example of a COOP main-loop gadget (ML-G).

As shown in Listing 1, the `removeElement` method will loop over an array, namely the field `L` and invoke the virtual method `remove` on every object stored in the field `L`. From an adversarial COOP perspective, this means that the attacker can inject arbitrary malicious objects and store them in the corresponding `L` field. Once she can invoke the `removeElement` method, the attack will be launched.

More advanced variants of COOP relax this requirement for a container object holding references to other objects. Crane et al., for example, describe *Recursive-* and *Unrolled COOP* variants that allow different patterns of repetition [18]. By applying control-flow integrity, valid control-flow transfers can be restricted to the program’s call-graph. Chen et al. demonstrates that COOP can still succeed despite this constrain [14]

2.3 Execute-Only Memory (XOM)

Machine code in the `text` section of a program usually possesses read *and* execute privileges. The read privilege is required to process inlined data, such as jump tables for `switch` statements. But the read privilege requirement is not *strict*. The only essential privilege for code is the ability to execute. Inlined data must then move to another section with read privileges.

The principle of least privilege—a core tenet of computer security policies—prescribes that reducing privileges improves security. Thus, in the 60s the Multics project already supported execute-only memory [16]. Over the past decade, the idea of execute-only memory saw a revival [4, 45, 17, 18, 22, 6]. The revival was due to advanced, sophisticated multi-stage attacks that used memory leaks to (i) read a processes code layout, and then to (ii) relocate a generic attack to the specific code layout used by a program. These specific code layouts were derived from an active research area called "software diversity," and complementing existing methods with execute-only memory begot the new class of defenses called *leakage-resilient diversity*.

2.4 Message Integrity Through MACs

To verify the authenticity and integrity of a message sent over an untrusted medium, people use so-called *message-authentication codes*, MACs for short. Both sender and receiver agree on a message authentication code (MAC) algorithm, based on a shared secret key k. Then, the sender computes the MAC checksum, also known as tag t, for every message m: $t = MAC(m, k)$ and sends this tag t along with the message. At the receiving end, we recompute the tag t' for the received message m': $t' = MAC(m', k)$. Then, by comparing both tags t and t' for equality, we verify the message m's integrity. Since the MAC algorithm is based on a secret key k, only shared between sender and receiver, third-parties cannot compute valid tags. Typically, secure MAC algorithms are based on cryptographic keyed-hash functions.

Counterfeit-object-oriented programming exploits the fact that control data, such as `vptrs` are mixed with non-control data. Similar to buffer overflows, mixing both types of data proves to be a security problem when adversaries inject malicious objects.

3 Related Work

Due to the severity of counterfeit-object-oriented programming as an attack vector, a variety of defenses [24, 18, 38, 55, 5, 14] has been proposed. Prior work, thus, considers multiple different design criteria. These design criteria include: software-only [18, 5] vs hardware-based [29, 52], hardening applied to binaries [39, 54, 21, 20] vs software-only, differences w.r.t. protected program parts (such as, protecting `vtables`, `vtable`-pointers, or dynamic dispatch). Due to these differences, giving an exhaustive treatment is in direct conflict within traditional scope restrictions. We therefore focus on the most directly related work, and skip, e.g., prior work dealing with securing C++ programs without source code access.

Most closely related to Hobbit is CFIXX, which uses Intel's discontinued MPX extension to protect `vptrs` [10]. At its core, CFIXX separates `vptrs` from `vtables` and stores them into a dedicated memory area protected by MPX. In 2022, Xie et al. demonstrated a CFIXX version building on Intel's Control-Flow Enforcement Technology (CET) [52]. Recently, many defenses proposed the use of Intel's MPK extension. Unfortunately, using MPK is not compositional: If an application uses MPK itself, it cannot share its MPK use with any other component, such as a defense.

Compared to Hobbit, CFIXX highlights the need for a software-only approach that does not require specific hardware extensions beyond extended-page table support to enable execute-only memory.

CCFI, short for cryptographically-secured control-flow integrity, is another closely related defense—not specifically aimed at preventing COOP attacks, but providing comprehensive protection against essentially all forms of control-flow hijacking [29]. CCFI pioneers the use of MACs to protect code pointers. Unfortunately, to secure the keys from leaking, the system proposed to reserve vector registers (i.e., SSE's `xmm` registers), thus slowing down application relying on their use, such as media en- or decoders.

Compared to Hobbit, CCFI highlights the need to preserve performance characteristics of programs, primarily by finding alternatives to protect secret keys that do not result in prohibitive performance impacts.

Hardware-based approaches are inextricably bound to the hardware mechanism and thus prone to sun-setting, as in the case of Intel's MPX instructions, or lack of compositionality, as in the case of MPK extensions.

Defenses based on cryptographic primitives often suffer from poor performance, e.g., by effectively blocking vector registers, and the security-prerequisite of having cryptographic primitives not spill data onto the stack.

4 Threat Model

COOP is a rather sophisticated attack and will, thus, often be a last resort for attackers. We assume, consequently, that proper defenses against simpler attacks, such as code injection, ROP [44, 40], and return-into-libc [19] are in place. Since Hobbit aims to prevent COOP attacks, we assume a strong threat model in line with previous work [43, 18, 29, 10].

In general, launching a COOP attack requires an attacker to hijack an initial object, including its virtual table pointers (vptrs) and data, and inject new counterfeit objects. To that end, an attacker needs to read or infer addresses of Virtual Tables (vtables) and write object-like data, including vptrs and other data, to specific memory regions. A variety of vulnerabilities provide such capabilities, including buffer overflows [36] and use-after-free vulnerabilities [49]. Although a restricted read- and write capability might suffice, we assume an attacker capable of reading arbitrary readable memory and writing to arbitrary writable memory.

Our system relies on W^X, marking memory either as writable or executable, but not both at the same time. Writing to code residing in executable memory or execute written data is not possible. Therefore, injecting new code or modifying existing code is not possible.

The attacker's arbitrary read capability renders defenses relying on secrets in readable memory ineffective. For example, protecting against overflowing into control data, such as return addresses or vptrs with (stack) canaries, is not effective. An attacker can easily read these values and embed them in her payload, or—assuming an arbitrary write capability—skip canaries at all. To mitigate this issue, we assume Execute-Only Memory (XOM), therefore, we consider values or functions in XOM as secret.

Finally, we assume an attacker with specific knowledge about the target program and

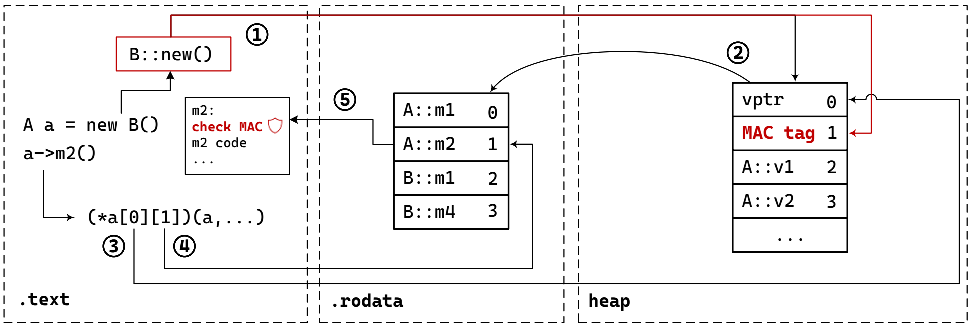


Figure 2 Hobbit changes to C++. ① constructors allocate objects, set `vptr` and field values *and* compute a MAC value, ②. Method calls are resolved as before, see Figure 1, but all method prologs now validate the MAC value, ⑤.

system. First, he has access to the target program's source code. Second, she is able to infer the base address of the initial object, and the addresses of virtual function gadgets (vfgadgets) located in C++ modules. Although COOP relies primarily on high-level C++ semantics, some vfgadgets rely on specific instructions or registers, e.g., vfgadgets for loading argument registers to pass arguments to other vfgadgets. An attacker requires at least partial knowledge of the binary layout to use some vfgadgets. Third, the attacker knows about the system's configuration, including deployed defenses, software versions, and hardware features.

5 Design Aspects of Hobbit

Hobbit is, broadly speaking, a defense that monitors and validates integrity. Whenever this integrity is violated by an adversary, we know that the program is under attack. A direct consequence of any integrity-protection mechanism also holds for Hobbit: we protect neither the injection, nor the modification of objects; subsequent method calls trying to *act* on maliciously-modified objects will detect integrity violations.

The integrity monitored by Hobbit is the object to `vptr` binding. One could just add a random value into an object and repeatedly validate its value. Since our threat model includes a powerful attacker with memory read capabilities, choosing a simple random value is insecure. Instead, Hobbit considers objects, more specifically `vptr`s, between constructors and methods as messages, and secures them by applying message-authentication codes.

The following sections provide an in-depth discussion of the relevant design aspects of Hobbit. Section 5.1 discusses C++ relevant aspects of object lifetime and changing the object layout to add the MAC tag. Sections 5.2 and 5.3 describe the benefits of using execute-only memory, and MAC-algorithm diversification. Section 5.4 lists possible locations for verifying signatures. Finally, we introduce the concept of gadget-directed optimization in Section 5.5.

5.1 C++ Object Lifetime and Layout

Objects in C++ live between construction and destruction, i.e., by constructors and destructors, respectively. Constructors instantiate an object by initializing, or assigning concrete values to its fields, which themselves are prescribed by their corresponding class definitions. Since a `vptr` is merely a field itself, at least from a run-time perspective, the constructor assigns the `vptr` of the called dynamic type. Destructors clean up object instances and, finally, free the allocated memory.

Hobbit changes the C++ object layout by adding a machine-word per `vptr` that holds the computed MAC tag (see Figure 2 ①). Besides requiring an extra word per `vptr`, such a change breaks the application binary interface (ABI), and we discuss the implications thereof in Section 6.

5.2 Message-Authentication Codes and Execute-Only Memory

In Hobbit, we consider `vptr`s as messages sent from constructors (see Figure 2 ①) and received by virtual methods (see Figure 2 ⑤). The key security aspect of MAC functions is the shared-secret key between senders and receivers. If an attacker retrieves this secret key, she can craft valid signatures for malicious messages, thus violating the authenticity property of sent messages. To prevent leakage of this shared-secret key, Hobbit piggybacks on execute-only memory's leakage-resilience property.

Execute-only memory means that the adversary is precluded from reading code memory. As a result, we can hide privileged information directly in code memory. Hobbit hides two privileged pieces of information there: (i) keys as intermediate constants, and (ii) MAC algorithm implementations. Hiding implementations from adversaries forces them to guess, thus further frustrating attacks.

MAC algorithm parameters, too, are important for security. Consider the following parameterization to compute object-`vptr` tags:

$$t = \mathrm{MAC}(\texttt{vptr} \oplus r) \tag{1}$$

Although we include a random parameter r to the MAC computation, our attacker can use their memory-read primitive to read an object—including its `vptr` and the corresponding tags—and, use it later on during an attack at a different location. Such a staged attack is called a "replay" attack. To counter these replay attacks, we need to add the `vptr` location to the computation:

$$t = \mathrm{MAC}(\texttt{vptr} \oplus \&\texttt{vptr} \oplus r) \tag{2}$$

By making MAC tags location-dependent, the attacker cannot trivially replay the object layout she read at a different location.

Prior defenses reserve registers to hold the key and exclude them from register allocation [29, 37]. Since the compiler then never allocates these registers, the key stored therein is considered safe from attackers. Although simple, this solution suffers from two drawbacks. First, reserving registers increases register pressure, which is particularly problematic on architectures with few registers, such as x86. Second, whether a key stored in registers is actually safe, depends on additional measures and precautions for context switches. Through its use of execute-only memory, Hobbit bypasses these shortcomings.

5.3 Class-Hierarchy-Driven Seed Randomization

By using just a single random parameter r in our MAC tag computation, the adversary can bypass Hobbit, once he identifies both the secret MAC algorithm and the value of r. Hobbit counters this problem by using as many random parameters r as possible. In theory, different random parameters r can be randomly assigned across an application. In practice, however, we need to preserve C++ semantics across type-compatible call-sites. A conservative way to ensure semantics preservation is to map a single random parameter r to a subgraph in the class hierarchy graph (see Section 6.4). A more aggressive way would be to factor in run-time information, e.g., through profiling.

Due to this additional security mechanism, we can also loosen the strength requirements for our MAC algorithm. By choosing small, but efficient pseudo hash functions, such as `moremur-hash` [30], HOBBIT users favor performance over security, and vice versa. Since MAC algorithm implementations are protected by execute-only memory (see Section 2.3), the perceptible loss of security is minimal.

HOBBIT supports a wide variety of MAC hashing algorithms, such as `blake3`, `highwayhash`, `xxhashct`, `moremur`, and `moremur-random`.

5.4 Validating MAC Tags

HOBBIT recomputes and validates tags stored in objects in function prologs of virtual functions (see Figure 2, ⑤). Although an attacker can inject malicious objects, HOBBIT will detect tampering with a tag *after* resolving the dynamic type, but *before* executing the actual method body. Alternatively, HOBBIT can also validate tags already at virtual call sites, but this implies embedding MAC hash computation into every call site, thus increasing the amount of machine instructions for each call site. Depending on the chosen MAC function implementation (e.g., inlined), these additional machine instructions might result in a considerable binary size increase.

In C++, most compilers use `vptrs` for other run-time related features besides dynamic dispatch. The use of run-time type information (RTTI), for example, requires loading the `rtti` pointer from the `vtable`. Similarly, dynamic casts use information stored in `vtables`, such as offsets to access/identify sub-objects for multiple inheritance. Although HOBBIT could validate tags in these cases, too, we choose to focus protection on dynamic dispatch, which is the key objective for COOP attacks.

5.5 Gadget-Directed Optimization

For performance-critical systems, such as real-time applications, HOBBIT can relax security and optimize for speed. Since COOP relies on special gadgets for dispatching other gadgets, we can embed integrity checks only in methods acting as such gadgets. To prevent attackers from executing Main Loop Virtual Function Gadgets (ML-Gs), HOBBIT can perform static analysis on source code to identify methods iterating over a collection of objects and calling virtual functions on them (see Section 6.5.)

HOBBIT could also analyze binaries to identify gadgets relying on binary instructions. Muntean et al., for example, created a tool for identifying gadgets and automating a COOP attack [32]. In general, identifying all gadgets is difficult and since variants of COOP exist, the resulting defense may not be complete [18, 14].

6 Hobbit Implementation

We implemented our prototype of HOBBIT as compile-time transformations on top of LLVM/Clang 17.0.3 [15] for the `x86_64 Linux` platform and `Itanium ABI` [23]. Most researchers implement their prototypes as passes in LLVM that operate on and modify the LLVM specific intermediate representation, short `LLVM IR`. However, we implemented most parts of HOBBIT in Clang, since compilation is a lossy transformation and high-level C++

information, e.g., virtual methods and their callsites, are not—at least without complex analysis—available in LLVM IR.

First, Hobbit extends the object layout to reserve space for the newly introduced MAC tag fields. After reserving space for MAC tags, we add instructions for computing and storing MAC tags in objects to constructors. For the final part of the `vptr` validation, we implement MAC tag checks in virtual methods. In Section 6.3 we describe our different MAC function implementations, Section 6.4 shows Hobbit's diversification implementation, and Section 6.5 demonstrates a prototype of our Gadget-directed Optimization. Section 6.6 lists the limitations of our prototype implementation of Hobbit.

6.1 Extending Object Layouts

Extending the object layout requires us to change the size of objects in a special data structure called `RecordLayouts`. Clang uses the type `CXXRecordDecl` to represent C++ structs, unions, and classes. `RecordLayouts` store information about fields, their offsets, paddings, and lengths, (virtual) bases, and other layout-related information. Since Hobbit introduces a new MAC tag field, we have to increase the size of the layout accordingly. On `x64` systems, pointers are eight byte long. Therefore, we add eight bytes to the (data-) layout size for dynamic `CXXRecordDecls` that do not inherit `vptr` (and consequently the MAC tag field) from a parent class in `AST/RecordLayoutBuilder` (`ItaniumRecordLayoutBuilder::LayoutNonVirtualBases`). Later, during the lowering of records, we add the field information for our MAC tag field, right after the `vptr` (see Listing 2).

```
void CGRecordLowering::accumulateVPtrs() {
  if (Layout.hasOwnVFPtr()) {
    auto vfptr = ...;
    Members.push_back(vfptr);
    auto HobbitMACField = MemberInfo(getSize(vfptr.Data),
                                     MemberInfo::Field,
                                     getIntNType(64));
    Members.push_back(HobbitMACField);
  }
  ...
}
```

Listing 2 Add MAC tag field while lowering records.

Extending the object layout breaks the C++ ABI compatibility. By recompiling the entire toolchain, including a standard C++ library, we still can compile and run programs with our C++ ABI modifications. We encountered one error in the `libunwind` library regarding macro definitions for the size of `libunwind::UnwindCursor`. `libunwind::UnwindCursor` is a dynamic class, therefore, consists of a `vptr` and with Hobbit also a MAC tag field. To fix this error we have to account for the new tag field and thus add one to all macro definitions defining the constant `_LIBUNWIND_CURSOR_SIZE` in `__libunwind_config.h`. With this simple fix, Hobbit can compile even the largest C++ programs.

6.2 Computing and Validating MAC Tags

C++ programs adhering to the C++ standard create objects solely by calling constructors. Therefore, we decided to implement the MAC tag computation and storing of the results

```
 1 _ZN1BC2Ev:
 2   ...
 3   leaq    _ZTV1B(%rip), %rcx # load address of vtable
 4   addq    $16, %rcx          # add 2 qwords for 1st virt. function = vptr
 5   movq    %rcx, (%rax)       # store vptr at beginning of object
 6 # HOBBIT START #
 7   movq    (%rax), %rdx       # load vptr into rdx register
 8   movq    %rax, %rcx         # load this into rdx register
 9   xorq    %rdx, %rcx         # vptr xor this
10   movabsq $random, %rdx      # load secret value r to rdx
11   xorq    %rdx, %rcx         # xor secret value r = mac tag
12   movq    %rcx, 8(%rax)      # save mac tag to designated field
13   # Possible inlined hashing or call to compiler-rt hash function
14 # HOBBIT END #
15   ...
```

Listing 3 `x86_64 assembly` for an exemplary constructor of a dynamic class B emitted by Hobbit.

```
_ZN1A2m2Ev:
# start function prolog:
  # save callee-saved registers
  # set up stack for local variables
  # ...
# HOBBIT START #
  movq    (%rcx), %rdx       # load vptr to rdx
  movq    %rcx, %rax         # load this ptr to rax
  xorq    %rdx, %rax         # vptr xor this
  movabsq $random, %rdx      # load secret value r to rdx
  xorq    %rdx, %rax         # xor secret value r = mac tag'
  movq    8(%rcx), %rcx      # load saved mac tag
  cmpq    %rcx, %rax         # check if tag' = tag
  jne     .LBB4_2            # on mismatch jump to trap
  ... # actual function      # actual function body
.LBB4_2: # %MACMismatchBlock # block with trap for mac tag mismatch
  movl    $147, %edi         # store result code 147 to edi
  callq   exit@PLT           # exit(147) on mac tag mismatch
# HOBBIT END #
```

Listing 4 `x86_64 assembly` for an exemplary virtual method of a dynamic class B emitted by Hobbit.

in constructors. Constructors already perform the `vptr` initialization in a function called `CodeGenFunction::InitializeVTablePointer`. Likewise, Hobbit initializes the MAC tags right after `vptr` initialization. Listing 3 shows the resulting assembly code of a constructor compiled with Hobbit. A standard `clang` compiler emits the three assembly instructions (lines 3–5) initializing the `vptr` of an object of a class `B`. Since `_ZTV1B` points to the beginning of the `vtable`—the first two entries in the `vtable` are the offset-to-top and the RTTI pointer—

Name	MAC Function	Implementation
baseline	-	-
no-hash	none; only xor(vptr, &vptr, random_secret)	-
blake3	C implementation of BLAKE3	static lib
highwayhash	highwayhash	shared lib
xxhashct	compile-time implementation of xxhash	static lib
moremur	pseudo hash function based on moremur	inlined
moremur-random	diversified version (random parameter) of moremur	inlined

Table 1 Details of implemented MAC functions used for benchmarking.

the compiler adds 16 bytes to the `vtable` such that the `vptr` points to the first virtual function and finally saves the `vptr` in the designated field at the beginning of the given object. The remaining instructions (lines 7–12) are emitted by HOBBIT and responsible for loading both `vptr` and `this` in registers, followed by the `xor` instruction. The `movabsq` instruction loads an immediate—the random secret r—to a register and `xor` it to the previous result. Finally, the `xor` result is written to the MAC tag field, 8 bytes after the `vptr`.

HOBBIT inserts MAC tag validation checks in virtual functions (see Listing 4). These validation checks protect against attackers calling virtual functions on objects with fake or altered `vptr`s, therefore mitigating COOP attacks. If HOBBIT should protect dynamic casts or RTTI access, we could insert MAC validation checks at those locations as well. To prevent the execution from virtual function bodies HOBBIT inserts the following instructions in `CodeGenFunction::StartFunction`:

1. We retrieve all `vptrs` for the current object.
2. For each `vptr`, we compute the MAC tag again.
3. For each `vptr`, we load the stored MAC tag value.
4. Then, we compare the computed and loaded MAC tag values.
5. If these tags match, we start executing the function body.
6. Otherwise, we detect an ongoing COOP attack and can launch counter-measures. In our prototype implementation, we simply exit the program with status 147.

Listings 3 and 4 show the resulting assembly code for both constructors and virtual methods of a class with 1 `vptr` without any hashing (`no-hash`).

6.3 MAC Function Implementations

We implemented different MAC functions in HOBBIT and extended the `baseline`, an unmodified Clang/LLVM 17.0.3. Table 1 shows the different hash implementations for the MAC function. The simplest approach is `no-hash` (as shown in Listings 3 and 4) that uses the identity function as MAC in Equation (2). Therefore, tag t is the unhashed result of the `xor` operations.

In contrast, `moremur` [30] implements a pseudo-hash function as MAC. These pseudo hash functions should be small, such that HOBBIT inlines these hash functions in both constructors and virtual functions. With XOM, immediate values used in such hash functions are resistant to leakage and can thus be considered secret. Section 6.4 describe `moremur-random`, a diversified implementation variant of `moremur`.

We implemented the remaining MAC functions, all including larger and more complex hash functions, as compiler run-time libraries, short `compiler-rt`. LLVM provides and links these libraries for run-time support in compiled binaries. We implemented different versions

```
...                            # preceding instructions from Listing 3
movabsq $random, %rax          # load random value to rax
xorq    %rax, %rdi             # xor random value = mac tag
callq   coop_hash@PLT          # call to compiler-rt hash function
movq    %rax, %rcx             # store result of coop_hash to rcx
movq    -16(%rbp), %rax        # reload this pointer
movq    %rcx, 8(%rax)          # save mac tag to designated field
...
```

Listing 5 Constructor calling a hash function in `rt-lib`.

of such a `compiler-rt` for the remaining MAC variants `blake3` [7], `highwayhash` [3], and `xxhashct` [53]. Hobbit links the `compiler-rt` libraries for `blake3` and `xxhashct` statically to the program under compilation. `Highwayhash`, in contrast, is dynamically linked as a shared library.

With run-time hashing support enabled, Hobbit simply inserts a call to the hash function located in the run-time library, according to Equation (2). Listing 5 shows the resulting instructions. After the initial `xor` instructions, the result is passed as an argument to the `coop_hash` function. The function `coop_hash` computes a hash according to the chosen hash function (Table 1), namely `blake3`, `highwayhash`, or `xxhashct`. Finally, after loading the `this` pointer again, the returned result is stored in the designed MAC tag field.

6.4 Class-Hierarchy-Driven Seed Randomization

In its current implementation, the random parameter r of Equation (2) is fixed over the whole program. We implemented a naive diversification approach diversifying this random parameter. Ideally, we would choose a different parameter for each class, however, due to the polymorphic nature of C++, the diversification degree is limited. We create MAC tags in constructors and validate them in virtual functions, therefore, both MAC functions must use the same random parameter. With subtyping, methods must be callable for different classes, according to the inheritance graph. Therefore, our diversified implementation chooses a random parameter for each weakly connected subcomponent of the inheritance graph. The inheritance graph is, in fact, a directed acyclic graph[1], since C++ has the concept of multi-inheritance, hence the famous *diamond problem.*

We implemented `moremur-random` in the following steps:

1. In an initial compilation step, Hobbit outputs all classnames with the corresponding (virtual-) bases.
2. We implemented a Python script that constructs the inheritance DAG.
3. Our script assigns each weak component[2] a different random parameter r.
4. Hobbit then use this class assignment to diversify the MAC tag computation.

By enabling `link-time optimization`, we could implement the inheritance graph analysis and the diversification assignment in Clang/LLVM.

---

[1] Not a tree, as one would expect.

[2] All connected subgraphs, also called *components*, ignoring the direction of edges.

	EPYC 7H12	**i7-8559U**	**Ryzen 9 5900X**
Processor	AMD EPYC 7H12	Intel 8559U	AMD Ryzen 9 5900X
RAM	1 TB DDR4	64 GB DDR4	64 GB DDR4
OS	Debian 12	Debian 12	Ubuntu 22.04.4 LTS
Kernel	6.1.0-16-amd64	6.1.0-16-amd64	6.5.0-27-generic
gcc	12.2.0	12.2.0	11.4.0
glibc	2.36	2.36	2.35
linker	gold (2.38)	gold (2.38)	GNU ld (2.38)

Table 2 Benchmark system configuration.

6.5 Gadget-Directed Optimization

We implemented a simple gadget-directed optimization that identifies simple main-loop gadgets. With this optimization enabled, HOBBIT performs a static analysis to identify potential main-loop gadgets. Our naive analysis checks whether a virtual method belongs to a class declaring any fields of C++ standard container type [11], either directly or indirectly, by inheriting from classes with such fields. This prototype gadget-directed optimization only identifies simple main-loop gadgets, but fails to identify other forms of dispatcher gadgets, serving as a main-loop gadget [43, 18]. Other dispatcher gadgets include `recursive gadgets`, `unrolled COOP`, or iterators over linked lists. HOBBIT could use COOP exploit automation frameworks, such as `iTOP`, to identify additional gadgets and feed them into our gadget-directed optimization [32].

6.6 Limitations

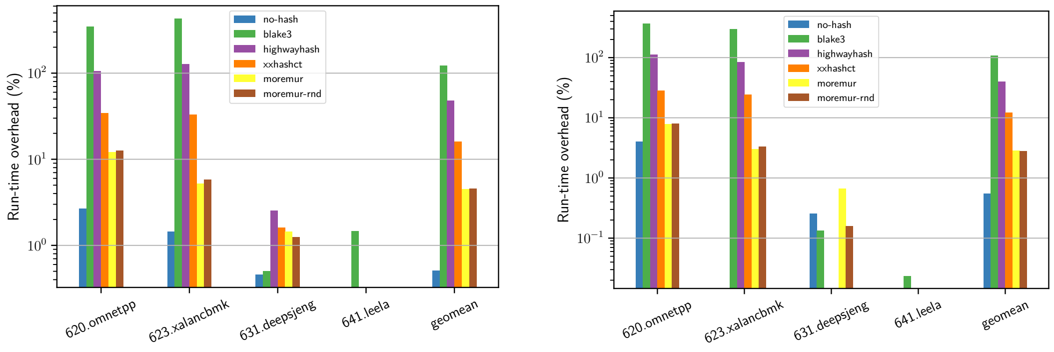
HOBBIT does not protect RTTI objects. RTTI objects are dynamic types, but not created by calling constructors at run-time. Instead, Clang initializes RTTI objects during compilation, therefore, HOBBIT does not compute and store MAC tags for such objects. At load-time, `vtables` and RTTI objects alike are loaded into `.rodata`. However, protecting RTTI objects is still possible but requires extra effort. We could, for example, create initialization code similar to our MAC tag initialization in constructors and call this RTTI object initializer when the address of both `vtables` and RTTI objects is known, at load-time. Since HOBBIT does not create MAC tags for RTTI objects, we do not emit integrity checks in virtual functions belonging to RTTI classes.

7 Evaluation

We present the evaluation of our prototype implementation of HOBBIT. In Section 7.1, we describe the machines used for our evaluation. Sections 7.2–7.4 show the performance evaluation, including measurements of run-time, memory-usage, and code-size. We evaluate our implemented prototype of gadget-directed optimization in Section 7.5. In Section 7.6, we evaluate the scalability of HOBBIT by compiling real-world applications with HOBBIT. Finally, Section 7.7 shows the evaluation of the class-hierarchy-driven seed randomization.

7.1 System Configuration

We perform our evaluation of HOBBIT on three different machines listed in Table 2.



(a) EPYC 7H12. (b) i7-8559U.

Figure 3 Run-time overhead introduced by Hobbit for C++ benchmarks of the SPECspeed™ 2017 Integer test suite, relative to baseline on log-scale.

We used machines EPYC 7H12 and i7-8559U for the performance evaluation in Section 7.2 and the gadget-directed optimization evaluation in Section 7.5. The scalability evaluation in Section 7.6 and the evaluation of the diversification statistics in Section 7.7 were done on Ryzen 9 5900X.

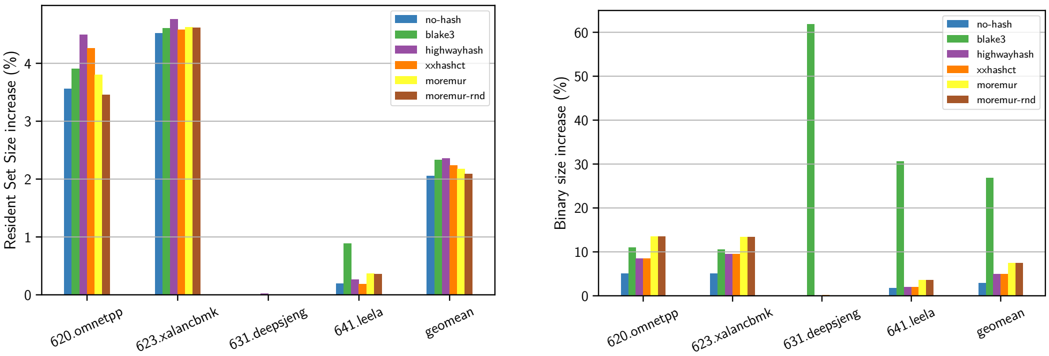
Our prototype of Hobbit is based on the LLVM/Clang version 17.0.3 (see Section 6), which we call baseline in the following evaluation. Since Hobbit breaks the C++ ABI, we have to build and use a custom-built version of the LLVM C++ standard library libc++ [27] (same as LLVM/Clang: 17.0.3). To improve comparability—although not strictly necessary—we build and use a custom-built libc++ for the baseline as well.

7.2 Performance

As common in performance evaluations, we evaluate the performance of Hobbit by building the SPEC CPU 2017 benchmark with our compiler modifications. In particular, since Hobbit only applies changes to C++ programs, we run the four C++ benchmarks of the SPECspeed™ 2017 Integer test suite, namely 620.omnetpp, 623.xalancbmk, 631.deepsjeng, and 641.leela. The remaining non-C++ benchmarks showed—as expected—no measurable overhead. As mentioned in Section 7.1, we use the custom-built libc++ instead of the bundled version of the Linux distribution. Each experiment compiles all relevant benchmarks and runs the compiled benchmark afterwards. We repeated each experiment 10× on EPYC 7H12 and 6× on i7-8559U and calculated the geometric mean over those repetitions.

Run-time, a key metric within SPEC, quantifies the time in seconds required for a benchmark to execute. Figure 3 shows the results for all evaluated MAC functions (see Table 1).

For the i7-8559U machine, the geometric mean overhead over all benchmarks, are 107.62% (`blake3`), 40.40% (`highwayhash`), 12.21% (`xxhashct`), 2.83% (`moremur`), and 2.80% (`moremur-random`). In comparison, on EPYC 7H12, the benchmarks show a higher performance impact over all benchmarks, namely 121.63% (`blake3`), 47.81% (`highwayhash`), 16.02% (`xxhashct`), 4.49% (`moremur`), and 4.54% (`moremur-random`). Both, 620.omnetpp and 623.xalancbmk, show the most performance impact on both machines. On i7-8559U, 620.omnetpp shows the highest run-time increase consistently for all benchmarked MAC functions. In contrast, on EPYC 7H12, we see a significantly higher run-time overhead on 623.xalancbmk for `blake3` and `highwayhash` compared to 620.omnetpp. The remaining



(a) Run-time memory increase.

(b) Binary code size increase.

Figure 4 Memory effects of HOBBIT for C++ benchmarks of the SPECspeed™ 2017 Integer suite, relative to baseline (EPYC 7H12).

hash functions (`xxhashct`, `moremur`, and `moremur-random`) on EPYC 7H12 show the same trend as on i7-8559U, namely, a higher performance overhead for 620.omnetpp rather than 623.xalancbmk.

We also evaluated a stripped down version that does not compute MAC tags to measure the minimum overhead (no-hash in Figure 3). On i7-8559U no-hash introduces a geometric mean overhead of 0.55%, with a maximum performance impact of 4.00% (620.omnetpp). In contrast to “correct” hash functions, the implementation of no-hash is 7.27% faster on EPYC 7H12 (overall 0.51%; 620.omnetpp 2.66%) when compared to i7-8559U.

7.3 Memory

Since HOBBIT extends object layouts, therefore, increases the size of objects, we are interested in the maximum `resident set size` (RSS). RSS is a metric indicating the memory usage of a process in RAM. Swapped memory does not count to RSS. By querying the `rusage` counters [41], our benchmarking environment measures the maximum RSS `maxrss`.

Figure 4 shows the benchmarking results for machines EPYC 7H12 and i7-8559U. On both machines, our benchmarks show an overall geometric `maxrss` overhead of 2.2% and 2.18%, respectively. We see the highest `maxrss` overhead for 623.xalancbmk (EPYC 7H12 4.64%, i7-8559U 4.65%). 620.omnetpp has a similar `maxrss` overhead (EPYC 7H12 3.99%, i7-8559U 3.88%), whereas HOBBIT has a low `maxrss` impact on 641.leela (EPYC 7H12 0.41%, i7-8559U 0.32%). For 631.deepsjeng, our defense does not increase the `maxrss` on neither machine at all.

7.4 Code Size

HOBBIT inserts instructions for creating and validating MAC tags and, for some MAC functions, links run-time libraries and creates function calls to these libraries. These additional instructions (and libraries) increase the binary size of compiled programs. To that end, we evaluate the binary size of each benchmark. Table 3 shows the binary sizes of the baseline benchmarks (see Table 3a) and the run-time hashing libraries (see Table 3b).

The binary size increase on both machines is identical and shown in Figure 4b. HOBBIT, in its `blake3` variant, introduces the highest geometric mean increase in binary size of 26.90% over all benchmarks, ranging from 10.50% for 623.xalancbmk up to 61.87% for 631.deepsjeng.

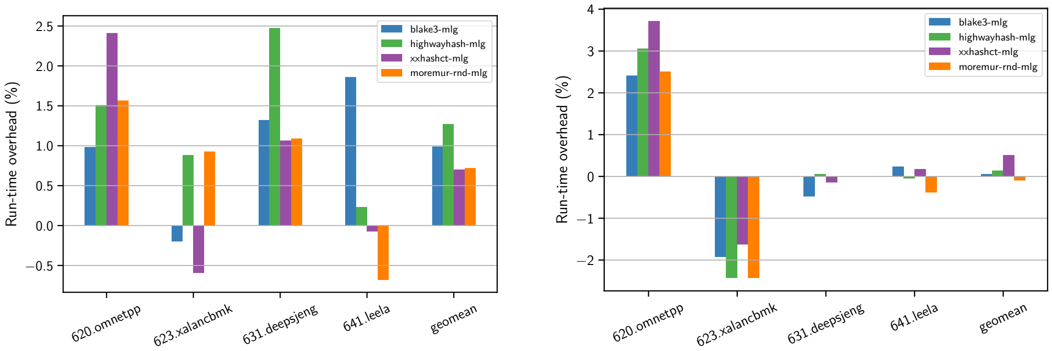
Name	Size in Bytes
620.omnetpp	2,915,320
623.xalancbmk	7,362,408
631.deepsjeng	118,120
641.leela	254,936

(a) Binary sizes of baseline benchmarks.

Name	Size in Bytes
blake3	90,618
highwayhash	15,816
xxhashct	1,874

(b) Binary sizes of hashing run-time libraries.

Table 3 Binary sizes of benchmarks and run-time libraries for both machines EPYC 7H12 and i7-8559U.



(a) EPYC 7H12.

(b) i7-8559U.

Figure 5 Reduction of performance impact through gadget-directed optimization.

`Blake3` is a big hashing library (see Table 3b). Since HOBBIT links `blake3` statically to the compiled program, the big library size, compared to small benchmarks as in 631.deepsjeng and 641.leela, contributes to the significant increase in the resulting hardened binary. On the other hand, for `highwayhash`, nearly 8.5× bigger than `xxhashct`, accounts for roughly the same binary size increase as `xxhashct`. The reason for this similar increase in binary size—despite a different library size itself—results from a different linkage. `Highwayhash` is dynamically linked, whereas `blake3` and `xxhashct` are statically linked, therefore, embedded in the binary. HOBBIT variants that inline MAC functions in constructors and virtual functions, namely `mormeur` and `moremur-random`, introduce the highest increase in binary size for 620.omnetpp (13.54%) and 623.xalancbmk (13.36%).

7.5 Gadget-Directed Optimization

We evaluated our naive implementation for the main-loop gadget analysis optimization (see Section 6.5), that only creates and validates MAC tags for classes having a standard C++ container field.

Although our gadget-directed optimization finds no main-loop gadgets for benchmarks 623.xalancbmk, 631.deepsjeng, and 641.leela, it finds 12 instances of classes having—directly or indirectly—at least one container-type field. HOBBIT inserts MAC tag integrity validation logic in 137 methods of these 12 classes.

Figure 5 shows the run-time overhead introduced by HOBBIT with gadget-directed optimization enabled.

Benchmark	blake3	highwayhash	xxhashct	moremur-random
Kraken 1.1 [25]	2.72%	0.70%	0.77%	−2.05%
MotionMark 1.3 [31]	14.64%	1.67%	1.87%	3.03%
Octane 2.0 [34]	53.54%	17.32%	2.83%	1.34%
Speedometer 2.1 [48]	161.74%	43.24%	7.85%	2.54%

Table 4 Performance impact on browser benchmarks.

7.6 Scalability

To evaluate the scalability of Hobbit, we compiled WebKit, a web browser engine consisting of millions of lines of C and C++ code (see Table 5). Specifically, we built the GTK version of Webkit, `WebKitGTK` [51], a full-featured port of WebKit for GTK-based Linux desktop systems. Although Hobbit breaks the C++ ABI through its object-layout extension, we only needed a single change to successfully compile WebKit, shown in Listing 6. Since `ScrollableArea` is a dynamic class, Hobbit inserts a field for the MAC tag, thus we have to add 8 to this `static_assert` to account for the increased object size.

```
#if CPU(ADDRESS64)
-static_assert(sizeof(ScrollableArea) == sizeof(
    SameSizeAsScrollableArea),
    "ScrollableArea should stay small");
+static_assert(sizeof(ScrollableArea) == sizeof(
    SameSizeAsScrollableArea) + 8,
    "ScrollableArea should stay small");
#endif
```

Listing 6 Fix required to compile WebKitGTK.

After the compilation, we evaluated the run-time overhead introduced by our defenses with the following browser benchmarks: (i) Kraken, (ii) MotionMark, (iii) Octane, and (iv) Speedometer.

As this evaluation requires a GUI, we performed the experiments on Ryzen 9 5900X. With only a terminal window opened, we started the `MiniBrowser`, a minimal browser based on `WebKitGTK`. After each benchmark execution, we closed the `MiniBrowser`, waited for ten seconds and repeated the experiment. In total, we executed each benchmark three times. Table 4 shows the geometric mean performance impact of our evaluation. Kraken measures the time needed to finish the benchmark, therefore, an induced overhead means an *increase* in run-time. In contrast, the other benchmarks measure *score points*, meaning that an induced overhead *decreases* the achieved score.

These real-world benchmark results confirm the results obtained from compute-intensive programs. Hobbit allows balancing security and performance, and we did not notice perceptible delays in daily browsing activities.

To further show that Hobbit scales to other real-world programs, we successfully compiled the following programs listed in Table 5. We included the version of the compiled programs as well as their C++ source lines of code (SLOC). The selected programs range from small web frameworks to fully fledged web browsers and compiler. For measuring SLOCs we used the tool `sloccount` [46].

Program	Description	Version	SLOC (C++)
crow	C++ Web framework	1.2.0	25,203
json	JSON library for C++	3.11.3	102,977
llvm	Collection of compiler tools	17.0.6	2,201,374
webkitgtk	GTK port of WebKit	2.41.1	4,444,590
620.omnetpp	SPECspeed®2017 Integer suite	SPEC CPU 2017	63,100
623.xalancbmk	SPECspeed®2017 Integer suite	SPEC CPU 2017	243,046
631.deepsjeng	SPECspeed®2017 Integer suite	SPEC CPU 2017	7,284
641.leela	SPECspeed®2017 Integer suite	SPEC CPU 2017	30,473

Table 5 Source lines of code (SLOC) of real-world programs compiled with Hobbit.

Top 5	libc++	omnetpp	xalanc	deepsjeng	leela	WebKitGTK
1.	78	193	442	78	78	3,916
2.	45	78	93	45	45	1,962
3.	27	52	78	27	27	1,541
4.	13	45	62	13	14	1,066
5.	12	34	49	12	13	427
Overall	**197**	**379**	**539**	**197**	**252**	**30,438**

Table 6 Top-5 and overall weakly connected component set size for libc++, C++ benchmarks of SPECspeed™ 2017 Integer, and WebKitGTK.

7.7 Class-Hierarchy-Driven Seed Randomization

We evaluated the number of diversified random parameters for our implementation from Section 6.4. Table 6 shows the Top-5 weakly connected components, that constitute the diversification unit. Each of these units is a set of classes for whom we must choose the same random parameter. All C++ benchmarks of SPECspeed™ 2017 Integer and WebKitGTK depend on libc++ and, thus, include and extends libc++'s inheritance graph. 631.deepsjeng does not introduce any new dynamic classes to the inheritance graph, whereas WebKitGTK adds $30,241$ new weakly connected components.

8 Discussion

We discuss and interpret the relevant findings of our evaluation.

8.1 Performance

Our performance evaluation shows that the performance impact depends primarily on the choice of the MAC algorithm. Although `blake3` offers the highest security, its performance impact, too, is the highest. To improve performance, Hobbit offers two complementary options. First, users can opt to use simpler MAC algorithms, such as `moremur`, which is more performance friendly. Second, users can apply our gadget-directed optimization to reduce performance impact of even the most expensive MAC algorithms.

Since we did not find any impact on large, real-world software, such as the WebKit browser, we argue that Hobbit can be used in a wide variety of contexts.

Exploit	LLVM	LLVM-CFI	Hobbit	Hobbit+LLVM-CFI	Hobbit-VFCS
FakeVT	✗	✓	✗	✓	✓
FakeVT-sig	✗	✓	✗	✓	✓
VTxchg	✗	✓	✓	✓	✓
VTxchg-hier	✗	✗	✓	✓	✓
COOP	✗	✗	✓	✓	✓

Table 7 Results of testing different `vtable` related attack building blocks against LLVM, LLVM CFI, and different configurations of HOBBIT.

8.2 Security

We compiled the `CFIXX-Suite` [12] with our Hobbit compiler. This exploit coverage test suite, created by Burow et al., demonstrates several scenarios for attacks on the dynamic dispatch mechanism [10]. Our security evaluation of Hobbit is shown in Table 7. Hobbit, in its initial version, only protects against scenarios 3–5 (namely `VTxchg`, `VTxchg-hier`, `COOP`), but fails to detect scenarios 1–2 (namely `FakeVT`, `FakeVT-sig`).

The initial Hobbit implementation prevents malicious execution of virtual function bodies by validating the integrity of `vptr`s in the function prologue. Since scenarios 1–2 insert fake `vtable`s that contain pointers to non-virtual functions, therefore unprotected by our defense, our prototype implementation does not prevent this form of attacks.

However, Hobbit is compatible and composable with other defenses such as LLVM CFI [28]. We compiled the exploit coverage test suite with Hobbit again, this time with `vcall sanitizer` enabled. To enable LLVM CFI, we provided the following compiler flags:

```
-fsanitize=cfi-vcall -flto -fuse-ld=lld -fvisibility=hidden
```

LLVM CFI succeeds in defending against fake `vtable` attacks and limits successful virtual calls to valid subtypes of the dispatched object's static type. Still, LLVM CFI fails to prevent an attacker from maliciously changing `vptr`s adhering to the type hierarchy or inserting fake objects without calling the appropriate constructor—the core principle of COOP. Combining Hobbit with LLVM CFI protects against all five exploit types evaluated in the exploit coverage test suite.

To account for situations where CFI cannot be used, we implemented an extension of Hobbit, namely Hobbit-VFCS. This Hobbit extension moves validation code from the function prologue of virtual functions to their call sites. Hobbit-VFCS validates `vptr`s *after* loading the `vptr` (Figure 2 ③), but *before* invoking the method call (Figure 2 ④). Emitting validation checks at each call site increases the binary size, but mitigates all five exploits. In future work, we can apply the same principle—checking the validity of `vptr`s immediately after loading—to protect other `vtable` related mechanisms, such as dynamic casts, too.

8.2.1 Balancing Performance and Security

Hobbit has, essentially, two orthogonal compile-time parameters: (i) hash function algorithm selection, and (ii) validation code granularity. By selecting a strong hash function, such as `blake3`, the overall security improves at the cost of performance. Conversely, selecting a more efficient hash function, such as `moremur`, decreases security and increases performance.

To offset the performance penalties, Hobbit offers users to parameterize the granularity of validation code insertion. Either all virtual functions or only COOP-relevant call sites are protected. By protecting all call sites, HOBBIT achieves the highest security at the potentially highest performance impact (i.e., by selecting an "expensive" hash function). Conversely, by selecting only the COOP-relevant call sites, HOBBIT reduces performance impact to a negligible level.

Although four different levels can be specified, we recommend the following settings in practice. A strong hash function, such as `blake3`, should be combined with COOP-relevant gadget granularity. A weak hash function, such as `moremur`, can be used to protect all virtual functions.

8.2.2 Uniformly Distributed Vtables

A method to perform cryptanalysis is to correlate input with output characteristics. Known-plaintext attacks are a form where the attacker knows the plaintext and infers a model from the outputs. In our model both inputs and outputs are either known or can be read directly through a memory-read primitive. The MAC algorithm used is hidden away effectively through execute-only memory. Yet, some of the input characteristics may allow attackers to launch a known-plaintext attack.

Consider, for example, that the attacker knows the addresses of `vtables` v_1, v_2, and v_3. Let's assume that although the addresses of these `vtables` v_i are different, their distances may remain constant. An adversary could, therefore, rely on such constant inter-table differences to infer properties about the concrete hash MAC algorithm used by HOBBIT.

Although our present implementation does *not* address this issue, we can achieve uniform distribution of inter-table differences by way of randomizing the order of emitting `vtables`. If this randomization proves to be insufficient, padding entries can be added in between emitted `vtables` to increase the entropy of `vtable` addresses.

9 Conclusions

HOBBIT presents an integrity-protection mechanism to thwart counterfeit-object-oriented programming attacks. At its core, this attack shares a symmetry to classical buffer overflows, in the sense that the underlying problem is the mixing of control with non-control data. For buffer overflows, this mix consists of keeping return addresses among stack frame data. For COOP attacks, this mix consists of keeping the `vptr` among object field data. By injecting malicious objects, the adversary can thus hijack control-flow and initiate illegitimate method calls.

To stop this type of whole-function code-reuse attack, HOBBIT changes the object layout to embed a tag value. This tag is computed by MAC functions that encode `vptr` information, `vptr` location, and a random secret. By leveraging execute-only memory, HOBBIT provides additional security. Due to complementary optimizations, users gain the ability to balance performance and security.

A comprehensive analysis provides evidence of both (i) configurable performance impact between 121.63% and 2.80% and (ii) scalability to multi-million lines of C and C++ code. At the same time, HOBBIT does not depend on MPX and does not inhibit performance by reserving registers. Without any hardware requirements, HOBBIT is applicable to embedded- and IOT devices.

References

1 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353, New York, New York, USA, 4 2005. ACM. `doi:10.1145/1102120.1102165`.

2 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 11 2009. `doi:10.1145/1609956.1609960`.

3 Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. Fast keyed hash/pseudo-random function using SIMD multiply and permute. *CoRR*, abs/1612.06257, 12 2016. URL: `http://arxiv.org/abs/1612.06257`, `arXiv:1612.06257`, `doi:10.48550/arXiv.1612.06257`.

4 Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1342–1353, New York, New York, USA, 2014. ACM. `doi:10.1145/2660267.2660378`.

5 Markus Bauer and Christian Rossow. Novt: Eliminating C++ virtual calls to mitigate vtable hijacking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 650–666. IEEE, 9 2021. `doi:10.1109/EuroSP51992.2021.00049`.

6 Felix Berlakovich and Stefan Brunthaler. R2C: aocr-resilient diversity with reactive and reflective camouflage. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 488–504, New York, NY, USA, 5 2023. ACM. `doi:10.1145/3552326.3587439`.

7 BLAKE3/c at master · BLAKE3-team/BLAKE3. URL: `https://github.com/BLAKE3-team/BLAKE3/tree/master/c`.

8 Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40, New York, New York, USA, 2011. ACM. `doi:10.1145/1966913.1966919`.

9 Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, 4 2017. `doi:10.1145/3054924`.

10 Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. CFIXX: object type integrity for C++. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, Reston, VA, 2 2018. The Internet Society. URL: `https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-2_Burow_paper.pdf`, `doi:10.14722/ndss.2018.23279`.

11 C++ Containers. URL: `https://cplusplus.com/reference/stl/`.

12 CFIXX Suite. URL: `https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite`.

13 Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 559–572, New York, New York, USA, 2010. ACM. `doi:10.1145/1866307.1866370`.

14 Kaixiang Chen, Chao Zhang, Tingting Yin, Xingman Chen, and Lei Zhao. Vscape: Assessing and escaping virtual call protections. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1719–1736. USENIX Association, 8 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/chen-kaixiang`.

15 Release LLVM 17.0.3 · llvm/llvm-project. URL: `https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.3`.

16 Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In Robert W. Rector, editor, *Proceedings of the 1965 fall joint computer conference, part I, AFIPS 1965 (Fall, part I), Las Vegas, Nevada, USA, November 30 - December 1, 1965*, pages 185–196. ACM, 11 1965. `doi:10.1145/1463891.1463912`.

17 Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, volume 2015-July, pages 763–780. IEEE Computer Society, 5 2015. `doi:10.1109/SP.2015.52`.

18 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 Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 243–255, New York, New York, USA, 2015. ACM. `doi:10.1145/2810103.2813682`.

19 Solar Designer. lpr LIBC RETURN exploit, 8 1997. URL: `https://insecure.org/sploits/linux.libc.return.lpr.sploit.html`.

20 Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for C++ binaries. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 140–154, New York, NY, USA, 4 2017. ACM. `doi:10.1145/3052973.3052976`.

21 Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 396–405, New York, New York, USA, 2014. ACM. `doi:10.1145/2664243.2664249`.

22 Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In Jaehong Park and Anna Cinzia Squicciarini, editors, *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*, pages 325–336. ACM, 3 2015. `doi:10.1145/2699026.2699107`.

23 Itanium C++ ABI. URL: `https://itanium-cxx-abi.github.io/cxx-abi/abi.html`.

24 Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL: `https://www.ndss-symposium.org/ndss2014/safedispatch-securing-c-virtual-calls-memory-corruption-attacks`, `doi:10.14722/ndss.2014.23287`.

25 Kraken JavaScript Benchmark (version 1.1). URL: `https://mozilla.github.io/krakenbenchmark.mozilla.org/index.html`.

26 Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 276–291. IEEE Computer Society, 5 2014. `doi:10.1109/SP.2014.25`.

27 "libc++" C++ Standard Library — libc++ documentation. URL: `https://libcxx.llvm.org/`.

28 LLVM: Control Flow Integrity. URL: `https://clang.llvm.org/docs/ControlFlowIntegrity.html`.

29 Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, volume 2015-October, pages 941–951. ACM, 10 2015. `doi:10.1145/2810103.2813676`.

30 Mostly mangling: Stronger, better, morer, Moremur; a better Murmur3-type mixer. URL: `https://mostlymangling.blogspot.com/2019/12/stronger-better-morer-moremur-better.html`.

31 MotionMark 1.0. URL: `https://browserbench.org/MotionMark/`.

32 Paul Muntean, Richard Viehoever, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. itop: Automating counterfeit object-oriented programming attacks. In Leyla Bilge and Tudor Dumitras, editors, *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*, pages 162–176. ACM, 10 2021. `doi:10.1145/3471621.3471847`.

33 Nergal. Advanced return-into-lib(c) exploits (PaX case study), 12 2001. URL: `http://phrack.org/issues/58/4.html#article`.

34 Octane 2.0 JavaScript Benchmark. URL: `https://chromium.github.io/octane/`.

35 Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 49–58, New York, New York, USA, 2010. ACM. `doi:10.1145/1920261.1920269`.

36 Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

37 Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz. Bytecode corruption attacks are real - and how to defend against them. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, volume 10885 of *Lecture Notes in Computer Science*, pages 326–348. Springer, 2018. `doi:10.1007/978-3-319-93411-2\_15`.

38 Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. VPS: excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching. In David M. Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 97–112, New York, NY, USA, 12 2019. ACM. `doi:10.1145/3359789.3359797`.

39 Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in COTS C++ binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, Reston, VA, 11 2015. The Internet Society. URL: `https://www.ndss-symposium.org/ndss2015/vfguard-strict-protection-virtual-function-calls-cots-c-binaries`, `doi:10.14722/ndss.2015.23297`.

40 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 3 2012. `doi:10.1145/2133375.2133377`.

41 getrusage(2) - Linux manual page. URL: `https://man7.org/linux/man-pages/man2/getrusage.2.html`.

42 AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. *J. Comput. Virol. Hacking Tech.*, 14(2):139–156, 5 2018. `doi:10.1007/s11416-017-0299-1`.

43 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 *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, volume 2015-July, pages 745–762. IEEE Computer Society, 5 2015. `doi:10.1109/SP.2015.51`.

44 Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561, New York, New York, USA, 2007. ACM. `doi:10.1145/1315245.1315313`.

45 Zhuojia Shen, Komail Dharsee, and John Criswell. Fast execute-only memory for embedded systems. In *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*, pages 7–14. IEEE, 9 2020. `doi:10.1109/SecDev45635.2020.00017`.

46 SLOCCount. URL: `https://dwheeler.com/sloccount/`.

47 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 *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 574–588. IEEE Computer Society, 5 2013. `doi:10.1109/SP.2013.45`.

48 Speedometer 2.1. URL: `https://browserbench.org/Speedometer2.1/`.

49 Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. Eternal war in memory. *IEEE Secur. Priv.*, 12(3):45–53, 5 2014. `doi:10.1109/MSP.2014.44`.

50 Arjan Van De Ven. New security enhancements in red hat enterprise linux v.3, update 3, 2004. URL: `https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf`.

51 WebKit/WebKit at webkitgtk-2.41.1. URL: `https://github.com/WebKit/WebKit/tree/webkitgtk-2.41.1`.

52 Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: retrofitting intel CET for generic and efficient intra-process memory isolation. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, CCS '22, pages 2989–3002, New York, NY, USA, 2022. ACM. `doi:10.1145/3548606.3559344`.

53 ekpyron/xxhashct: Compile time implementation of the 64-bit xxhash algorithm as C++11 constexpr expression. URL: `https://github.com/ekpyron/xxhashct`.

54 Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, pages 8–11, Reston, VA, 2015. The Internet Society. URL: `https://www.ndss-symposium.org/ndss2015/vtint-protecting-virtual-function-tables-integrity`, `doi:10.14722/ndss.2015.23099`.

55 Chao Zhang, Dawn Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. Vtrust: Regaining trust on virtual calls. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, Reston, VA, 2016. The Internet Society. URL: `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/vtrust-regaining-trust-virtual-calls.pdf`, `doi:10.14722/ndss.2016.23164`.