Type-after-type

Van Der Kouwe, Erik; Kroes, Taddeus; Ouwehand, Chris; Bos, Herbert; Giuffrida, Cristiano

published in
ACSAC '18 Proceedings of the 34th Annual Computer Security Applications Conference
2018

DOI (link to publisher)
10.1145/3274694.3274705

document version
Publisher's PDF, also known as Version of record

Type-After-Type: Practical and Complete Type-Safe Memory Reuse

Erik van der Kouwe
Leiden University
e.van.der.kouwe@liacs.leidenuniv.nl

Taddeus Kroes
Vrije Universiteit Amsterdam
t.kroes@vu.nl

Chris Ouwehand
Vrije Universiteit Amsterdam
palaga@gmail.com

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

Temporal memory errors, such as use-after-free bugs, are increasingly popular among attackers and their exploitation is hard to stop efficiently using current techniques. We present a new design, called Type-After-Type, which builds on abstractions in production allocators to provide complete temporal type safety for C/C++ programs—ensuring that memory reuse is always type safe—and efficiently hinder temporal memory attacks. Type-After-Type uses static analysis to determine the types of all heap and stack allocations, and replaces regular allocations with typed allocations that never reuse memory previously used by other types. On the heap, Type-After-Type splits available memory into separate pools for each type. For the stack, Type-After-Type efficiently implements type-safe memory reuse for the first time, pushing variables on separate stacks according to their types, unless they are provably safe (e.g., their address is not taken), in which case they are zero-initialized and kept on a special stack. In our evaluation, we show that Type-After-Type stops a variety of real-world temporal memory attacks and on SPEC CPU2006 incurs a performance overhead of 4.3% and a memory overhead of 17.4% (geomean).

CCS CONCEPTS

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

KEYWORDS

Use-after-free; uninitialized read; LLVM; computer systems; defense

ACM Reference Format:
Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3274694.3274705

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274705

1 INTRODUCTION

Temporal errors allow attackers to abuse the reuse of memory over time and rank among the most popular vulnerabilities in today's exploits. A typical example is use-after-free, where an attacker can use a pointer to a deallocated block of memory to access reallocated memory (or vice versa). Current defenses are expensive and/or limited in either scope or effectiveness (e.g., protecting only the heap or vulnerable to memory massaging). We describe a new production-ready solution which guarantees temporal type safety to mitigate most such attacks at very low overhead (just 4.3% on SPEC CPU2006 to protect both the stack and the heap).

Almost all temporal memory attacks exploit the lack of type safety in memory reuse. For instance, a use-after-free vulnerability in a C++ program may lead to VTable hijacking [37], because attackers can massage the heap to reuse the memory of a previously freed object for an object of a different type. If the program erroneously references the original object, it executes a method controlled by the attacker. Likewise, common uninitialized read attacks exploit memory reuse to leak sensitive information, such as return addresses stored on the stack [27]. Temporal type safety prevents such attacks by preventing reuse of memory for different types.

Defenses. Compared to spatial errors (e.g., buffer overflows), which can be mitigated by checking bounds [20, 21], temporal errors are much harder to stop. For a complete defense, we need to track every pointer in memory and incur high overheads (over 40% in prior solutions [8, 24, 42, 45]). A cheaper alternative is to probabilistically prevent memory reuse [4, 32], but such approaches can be easily bypassed [24].

A more promising and practical mitigation of temporal errors is the enforcement of type-safe memory reuse [2]. While this does not prevent reuse-based memory accesses by an attacker altogether, it ensures that such accesses are still type-safe, greatly constraining the attacker and preventing attacks against temporal vulnerabilities. However, current approaches [2] infer allocation types at runtime, reducing accuracy and efficiency. More importantly, they protect only the heap, leaving the stack vulnerable. The reason is that the stack is notoriously performance-sensitive and extremely challenging to protect without incurring high overhead [27]. Unfortunately, stack-based temporal attacks are popular and, for instance, Microsoft reports that, over the last ten years, 52% of the uninitialized read vulnerabilities on its platform concerned the stack [28].

Efficient Temporal Type Safety. We present Type-After-Type, which efficiently protects unmodified production programs against type-unsafe exploitation of temporal memory errors. Type-After-Type is the first system to enforce *complete* type-safe memory reuse, that is (i) on *both* stack and heap and (ii) mitigating *both* use-after-free and uninitialized reads. To determine the type of stack and heap memory allocations, Type-After-Type uses static analysis. While the type is explicit for stack allocations, on the heap we trace backward to determine how the allocation size is computed and forward to determine how the allocated memory will be used. We show that these approaches are complementary and together can find the heap allocation type in most cases. In the few remaining cases, even inaccurate type inference may result in type-unsafe reuse but only for the same callsite. This results in slightly lower security but never breaks programs, a key property for a practical defense.

Aiming for practical deployment in production, Type-After-Type supports common production allocators such as SafeStack [22] and tcmalloc [12] (another key difference with prior efforts [2]). In detail, our system replaces regular heap and stack allocations with typed allocations that never reuse memory for a different type. On the stack, we determine which variables can be used in unsafe ways and create separate stacks with their own stack pointers for each unsafe variable type. On the heap, we maintain separate type-based pools to prevent an allocation of one type from reusing memory that has previously been allocated as another type, effectively thwarting VTable hijacking and similar widespread temporal attacks. Together, these changes ensure that temporal memory errors cannot be exploited to achieve type-unsafe access.

Type-After-Type makes a number of important improvements compared to related systems. First of all, we combine protection of the stack and the heap to efficiently provide full protection. Compared to Cling [2], Type-After-Type protects the stack, improves performance on the heap, and improves type and allocation wrapper detection. SafeCode's type-safe heap reuse design relies on perfect static analysis yielding a complete call graph and precise points-to information, assumptions that are only realistic for small, embedded software [9]. In contrast, Type-After-Type does not rely on complete call graph or points-to analysis, allowing support for complex, real-world applications. Finally, existing efficient stack protections provide comparable performance to Type-After-Type's on both stack and heap, but even on the stack alone they cannot address all the temporal vulnerabilities [22, 25, 27].

Contributions. This paper makes the following contributions:

- Type-After-Type, a novel design ensuring complete temporal type safety for unmodified programs and seamless integration in production allocators.
- An evaluation which shows that we prevent attacks effectively and efficiently (4.3% overhead on SPEC CPU2006);
- A comparison of the effectiveness of different approaches to statically determine types of heap memory, important in a variety of other applications;
- An open-source prototype implementation is available at https://github.com/vusec/type-after-type, to serve as both a practical defense mechanism (with already interest from the industry), and a basis for further research.

2 BACKGROUND

In this section, we first discuss temporal memory errors and their exploitation. Then, we discuss the importance of type safety in unsafe languages such as C and C++, a property which Type-After-Type preserves even in the presence of temporal memory errors.

2.1 Use-after-free

A use-after-free vulnerability exists whenever a program allows a pointer to a previously deallocated block of memory (that is, a *dangling pointer*) to be dereferenced. After deallocation, the allocator eventually reuses memory for new objects. As a result, the dangling pointer and the new live pointer refer to the same block of memory. This means that writes to one object will corrupt the other and reads may either leak sensitive data from the other object or cause data from the other object to be misinterpreted.

As an example, we consider VTable hijacking, a common use-after-free exploitation technique in real-world attacks [37]. In C++, virtual method tables (VTables) help select an implementation of a virtual method at runtime. If a program creates a dangling object pointer, an attacker can craft an input that causes the program to allocate blocks of memory and store fake VTable pointers into them, a process known as heap [37] or stack [26] spraying (or massaging [24] in a more targeted form). Afterwards, calls to virtual methods use the fake VTable pointer supplied by the attacker, hijacking control flow. This example demonstrates how a simple and hard to find bug can lead to an attacker taking over control.

2.2 Uninitialized reads

Uninitialized reads are increasingly recognized as an important threat to application security [26]. C does not initialize allocated memory and, while C++ does automatically initialize class fields under some circumstances, the semantics are complex and there are still many cases where data is not automatically initialized [25]. Reading from uninitialized memory results in undefined behavior. Compilers can only detect this in trivial cases [25].

While undefined, attackers can generally predict the result of uninitialized reads. Old data remains in place when memory is reused. An attacker can use heap or stack spraying/massaging to control the uninitialized value read by the program.

2.3 Type safety

The attacks from the previous sections rely on the program interpreting one type as another. For both VTable hijacking and uninitialized reads, the attacker must spray/massage specific pointers over the stack or the heap. However, programs almost never accept pointers as input to the program. There is no legitimate use for this because pointers are meaningless outside the owning address space. Often, an attacker stores a pointer in a field that is not pointer-typed, but rather a type that is more sensible for user input such as an integer, a string or binary data. Temporal memory errors allow this data to be reinterpreted as a pointer type. This is a crucial step in allowing these attacks to succeed. Likewise, leaking a pointer to break ASLR requires the attacker to force the program to write its value to some output stream. Given that there is no legitimate use for the pointer outside the program, this only happens because

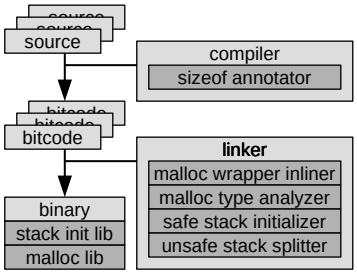

Figure 1: Overview of our framework; darker parts together constitute Type-After-Type

the program interprets the pointer as some other type. These examples show that type-unsafe memory reuse is critical in exploiting temporal memory errors, as also observed in prior work [2].

3 THREAT MODEL

We consider an attacker pursuing type-unsafe exploitation of temporal memory error vulnerabilities in a victim program. We assume the target program to be protected against other classes of vulnerabilities (e.g., buffer overflows) using orthogonal classes of defenses. Hence, we assume that such vulnerabilities cannot be used to access our metadata in the memory manager's data structures (to keep track of multiple typed heaps) and the thread local storage (to store the stack pointers for multiple typed stacks). This is justified since such vulnerabilities, when not adequately protected, can be already used for end-to-end attacks without the need to perform further violations of temporal type safety.

4 OVERVIEW

Type-After-Type takes the source code of an unmodified C/C++ program and compiles it into a hardened binary that enforces temporal type safety if the developer specifies a new compiler flag. The resulting binaries do not behave differently under normal circumstances, while modified memory management of both the heap and the stacks ensure that an attacker cannot break type safety even in the face of temporal memory errors.

Figure 1 shows the structure of our system. First, the build system compiles source files into object files using the LLVM compiler's intermediate representation (also known as "bitcode"). This is possible by simply using the `-flto` clang/llvm compiler flag. We add a small plugin in the compiler to prevent `sizeof` type information from being lost in the translation to bitcode. After the linker combines these object files into a single program, it runs the four plugin passes that are part of Type-After-Type. These passes perform static analyses and use the results to modify the generated code to support temporal type safety. Finally, the linker links two libraries supplied by Type-After-Type to the final binary to support the code generated by the passes.

The first two compiler passes alter heap allocations to make them type safe. The Malloc Wrapper Inliner conservatively detects memory allocation wrapper functions and inlines them into their callers to ensure the standard allocation functions are called in the function where the memory will actually be used. The Malloc Type Analyzer determine how each allocated block of memory is used and modifies the memory allocation call to specify the used type. This pass is supported by our Malloc Library, which implements type-based heap pools and exposes memory allocation functions with an extra parameter to specify the type of the allocation.

The remaining two passes protect the stack from type-unsafe memory reuse. The Safe Stack Initializer ensures that variables that are provably used only in a safe way (i.e., their address is never taken) are always initialized, so as to prevent exploitation of stack-based uninitialized read vulnerabilities. The Unsafe Stack Splitter deals with the remaining variables that may be used in unsafe ways by moving them away from the regular stack to separate stacks that only store variables of one type each. This pass is supported by the Stack Initialization Library, which implements type-based stack pools and exposes their stack pointers to our passes.

5 HEAP

On the heap, Type-After-Type ensures temporal type safety by splitting available memory into separate pools for each type. When an application frees memory, we do not return it to the general free list or the operating system but rather keep it available for future allocations of the same type. We override the default memory allocator to provide our own implementation of the *standard memory allocation functions*, namely `malloc`, `calloc`, `realloc`, and the `new` operator. However, in some cases a single function allocates memory for many different purposes, resulting in different types. To make these functions type-aware, we detect such allocation wrappers and inline them into their callers. This causes the allocation functions to be called in the context where the memory will be used, which simplifies the analysis.

5.1 Typed memory allocations

To make heap memory allocations type-aware, we scan the program for calls to the standard memory allocation functions. For each call, we detect the type of the allocated memory. We support two static type detection methods: cast-based analysis and `sizeof`-based analysis. If we can detect the type statically, we generate a (secure) 64-bit *type hash* that is always the same for equal types and (almost) always differs between different types. Otherwise, we use a *call site hash* unique to that specific call. This is a hash of the name of the function and the call's position in the function, computed at compile time. Finally, we modify the call to use a memory allocation function with an additional type parameter, which receives the hash value. In this way, we can make allocations type-aware without any run-time analysis.

Our alternative memory allocation functions are supplied by our custom Malloc Library which we link with the final binary, as shown in Figure 1. Each corresponds to one of the standard memory allocation functions with an additional type parameter. In our design we can in principle use any underlying (slab-like) memory allocator, if it is adapted to maintain separate pools for each type and to not release deallocated memory to the operating system. In particular, we demonstrate our design on top of the tcmalloc [12] high-performance heap allocator, as detailed later in Section 7. This strategy ensures that previously deallocated memory

is never reused for an allocation with a different type specified, introducing complete temporal type safety.

We cannot always determine the type of an allocation, for example because every approach yields either no result or a typeless pointer. In these cases we use a call site hash instead of the type hash. We use the same approach for a pointer to bytes, `char*` in C/C++, because this type is often used to store binary data without regard for types. In this case, deallocated memory is only reused for allocations from the same call site. As long as this call site always allocates the same type of memory, this provides maximum type safety at the cost of potentially increasing run-time and memory overhead. In the worst case, multiple types are allocated at the same call site. After our wrapper detection and inlining pass, (see Section 5.2) this is rare in practice. In this case it is impossible to guarantee complete temporal type safety, buy we still limit exploitability by having only allocations from the same call site share a memory pool. Moreover, our prototype does not reuse memory between size classes (see Section 7), providing protection between objects of different sizes as well. This is similar to the limitations of prior work [2], which (in all cases) suffers the same issue if there are either identical allocation sizes between the types or if the call stack is the same between the allocation of the different types.

To achieve type-safe memory reuse, newly allocated objects must never overlap partially with previously allocated ones, because that may cause different structure members with different type to end up at the same memory locations. Fortunately, this is not an issue for tcmalloc, which aligns allocations to their size class and prevents cross-class reuse. For other allocators, the same is achieved by adding the size to the type hash at run time.

`new` *operator type detection.* Whenever we encounter a memory-allocating `new` operator, we use the return type generated by the compiler to reliably determine the allocated type.

Cast-based type detection. For casts, we trace forward the return values from memory allocation functions, looking for typecasts. We consider both direct typecasts and indirect usage, for example if it is typecast multiple times. We perform the following propagations: for typecasts we consider how the result is used; for call arguments we consider how the callee uses the parameter; and for return statements we consider how callers use the return value. If we find at least one type other than `void*` and `char*` and all such types match, this is the final type. If there are inconsistent types, the result is inconclusive and we log a type conflict.

`sizeof`*-based type detection.* Our final approach uses the `sizeof` operator. This operator must be used in most heap allocations for portability. We trace backwards from the size of each call to a standard allocation function to determine whether it is computed from a `sizeof` operator. Although we cannot distinguish an array type from its element type, this does not threaten type safety because an array simply stores its elements next to each other.

Combined type detection. For allocations other than the `new` operator, we perform both forward tracing for cast operations and backward tracing for `sizeof` operators. If we find an array type, we use its element type for consistency with `sizeof`. This reduces the total number of types without affecting type safety. If we find multiple different types, we select the one we consider most interesting; that is, least likely to represent an intermediate step. In particular, we consider pointers to integers to be less interesting because they are often used as an intermediate type for data of other types such as structures. We consider pointers to function pointers less interesting because the C++ compiler often casts classes to this type to call virtual methods. We consider derived classes more interesting than their bases. If multiple types are equally interesting, we prioritize the `sizeof` approach because it has been deliberately specified by the programmer for the truly allocated type. If we find multiple equally interesting types from the same approach, we consider the type uncertain and use a call site hash and log this as a type conflict. Note that failure to detect the allocation type statically does not necessarily mean that the object is unprotected; the fallback to a call site hash still ensures type safety if the call always allocates the same type (i.e., if it is not in a wrapper that cannot be inlined).

5.2 Wrapper detection and inlining

Many programs use wrappers to allocate memory, for example for portability, error checking, and debugging. Wrappers hinder type analysis of memory allocations. We identify memory allocation wrappers and inline them into their callers. Programs implement their own wrappers with different functionality and call signatures, so there is no definite way to tell them apart from functions that allocate memory for their own use. Moreover, there may even be multiple levels of wrappers. Aiming for a practical mitigation, we introduce a simple, but effective, overapproximation.

Our algorithm to detect allocators starts by initializing the set A of allocator functions to the standard memory allocation functions. We then enumerate all calls to these functions. If a call to an allocator function occurs in a function that looks like it might be a wrapper, we consider the calling function to be a possible wrapper unless a `sizeof` expression is used to determine the allocated size. The underlying assumption is that the `sizeof` operator indicates that the function decides which type to allocate by itself, rather than getting an allocation size passed from its caller or other context such as object fields. We determine which functions look like wrappers by looking for functions that return pointers. We add the possible wrapper to A and repeat until we find no more new wrappers. Finally, we tell the compiler to inline all calls to the functions in A other than the standard allocation functions we started off with. By doing so, we allow the type analysis from the previous section to identify the right type as if the wrappers were not there.

As anticipated earlier, our approach may overestimate the number of wrappers. In C/C++, it is common to return a pointer even if the caller does not own the memory it points to, even in functions that allocate memory. However, inlining non-wrapper functions is harmless for security purposes. The only risk is introducing additional performance overhead, but our measurements show that the overhead from inlining wrappers is negligible (see Section 8.5).

In cases where we cannot detect a suitable type and use a call site hash, inlining results in the creation of multiple pools (one for each call site after inlining), effectively considering the call stack in determining which heap pool is used—but without costly run-time analysis. As a result, inlining increases security even in face of false positives. As such, we believe that it is best to err on the side of caution and inline everything that looks like a potential wrapper.

6 STACK

Memory management on the stack is rather different than on the heap. With just a single stack pointer per thread to specify which parts are allocated, one cannot keep track of multiple pools of free memory blocks as heap memory allocators do. Moreover, stack allocations and deallocations are very frequent and must therefore be much more efficient than those operations on the heap. Finally, the compiler is expected to automatically deallocate stack variables even if a function is exited prematurely by means of a `longjmp` call or an exception. All these factors make type-safe memory reuse on the stack much more challenging than on the heap, which also explains why this was not attempted by prior efforts [2]. On the other hand, stack variables always have their type specified at allocation time, which means no type analysis is needed.

One more distinction between stack and heap memory that can be used to improve performance of temporal type safety is that stack variables may be used without explicit pointers to them. For such variables, use-after-free errors are not possible. Therefore, as an optimization to reduce the number of type pools, we can split up the stack into safe and unsafe parts similar to what SafeStack [22] does for a different purpose (buffer overflow mitigation).

6.1 Guaranteed initialization on the safe stack

The safe stack is the traditional type-unsafe stack and contains only variables that do not have their address taken as well as "invisible" state such as return addresses and spilled registers. Because these variables have mixed types, the stack requires alternative protection to prevent type-unsafe reuse. The only threat for safe stack variables is uninitialized reads, so our pass for such variables merely initializes them to zero. We rely on the compiler to eliminate spurious zeroing. Prior zero initialization strategies have shown that standard double-store elimination passes in modern compilers are not sufficient to eliminate the zero-initialization overhead on the stack (and even custom ones struggle) [27]. However, Type-After-Type requires the optimizer to only deal with safe local variables that do not have their address taken. In absence of pointer aliasing problems, these variables are now much easier to analyze than the general case, significantly improving the effectiveness of standard double-store elimination and with no need for custom (and less practical) optimizations to achieve good performance. Our approach allows for very low (if any) overhead for safe variables, which represent the vast majority of local variables in practice.

6.2 Typed unsafe stacks

We protect variables on the unsafe stack by creating a separate stack for each type. Although we reserve stack memory for each type, in practice the OS allocates no physical pages for parts that are never used. For each type, we create a stack pointer variable in the thread-local storage (TLS), allowing each thread to keep track of its own typed stacks. To allocate an unsafe stack variable, we subtract its size from the correct typed stack pointer and replace references to the variable with the result. When a function returns, we restore the stack pointers for all unsafe stacks it used. In cases where a function may be the target of a `longjmp` call or contains an exception handler, we restore all unsafe stacks to their original positions. While this is relatively expensive if there are many types, our experiments have shown that this is generally rare.

Our Stack Initialization Library (shown in Figure 1) initializes the typed stacks used by instrumented programs when the program starts or a new thread is created. It allocates space for the typed stacks and sets up initial stack pointers. Each typed stack is preceded by an inaccessible guard page, preventing the stacks from growing into each other or other memory objects. Finally, whenever a thread is destroyed, we free its stacks to prevent memory leakage.

Like on the heap, we prevent partial overlap between prior and current memory objects. In particular, we protect variable-length arrays by rounding the stack pointer to a common multiple of the stack alignment and the element size. Though this causes extra memory overhead, in practice we found that the element size is almost always a multiple of 8 bytes, which means at most just one extra array element is needed to safely pad array allocations.

7 IMPLEMENTATION

We have implemented a prototype of the Type-After-Type design described in Sections 5 and 6. Our system is based on the LLVM compiler framework [23]. We use link-time optimizations, which causes LLVM to link together programs in its intermediate representation (also known as "bitcode"). This allows our compiler passes to run on the full program rather than one source file at a time, which simplifies inter-procedural static analysis.

Our heap allocator prototype is based on the high-performance tcmalloc [12] memory allocator. This memory allocator is widely used in production, for example in Google products such as the Chrome browser [14]. On the front-end side, we have extended the API (that is, the allocation functions with names starting with `tc_`) with `tc_typed_` versions that take an additional parameter to specify the type hash. Legacy allocation functions simply call these new versions with the type hash set to zero. This allows uninstrumented (and therefore unprotected) libraries to be used with protected binaries. These unsafe versions are also used whenever our pass cannot statically determine whether a call targets a memory allocation function, which is the case for indirect calls. We modified the memory management back-end to create one central free list for each known type hash and to have each thread cache maintain per-type free lists. The use of per-type free lists ensures that we do not reuse freed memory for allocations of a different type. To store the free lists, we use a hash table in both cases, indexed by the type hash. In addition to the added protection, tcmalloc's design also prevents memory reuse between different allocation size classes. This gives extra security in cases where the type cannot be detected; if the same call site allocates multiple types that fall in different size classes, memory reuse is still safe.

Our typed stacks are based on the SafeStack [22] pass included in the LLVM compiler. We modified this pass to create multiple unsafe stacks, each with their own entry in the thread local storage (TLS) to keep track of the stack pointer. We also create two additional TLS variables, placed before the first and after the last stack pointer. This allows our static library to initialize all stack pointers without knowing exactly which typed stacks exist.

We believe our Type-After-Type prototype offers substantial security improvements with practical, evolutionary changes to

CVE	Type	Source	Mitigated
2007-1521	Use-after-free	FreeSentry	Completely
2007-1522	Use-after-free	FreeSentry	Completely
2007-1711	Use-after-free	FreeSentry	Completely
2009-0749	Use-after-free	FreeSentry	Completely
2010-2939	Use-after-free	DangSan	Completely
2011-0065	Use-after-free	VTPin	Completely
2012-0469	Use-after-free	VTPin	Completely
2013-0912	Type confusion	Pwn2own	Exploit
2016-4342	Stack uninit read	SafeInit	Completely
2016-4486	Stack uninit read	SafeInit	Completely
2016-5337	Stack uninit read	SafeInit	Yes

Table 1: Vulnerabilities thwarted by Type-After-Type.

production allocators and we plan to seek mainline adoption in both tcmalloc [12] and SafeStack [22].

8 EVALUATION

To evaluate our Type-After-Type prototype, we performed a number of experiments. For our evaluation, we used Intel Xeon E5-2630 machines with 16 cores at 2.40 GHz with 64 GB of memory, running 64-bit CentOS Linux 7 with kernel version 3.10.0. In this section, we first consider Type-After-Type's security guarantees. Next, we discuss the effectiveness of our type and wrapper detection algorithms. Then, we consider Type-After-Type's performance and memory guarantees. Finally, we present results for a Firefox case study.

8.1 Security

We evaluated the effectiveness of Type-After-Type by examining a number of Common Vulnerabilities and Exposures (CVEs) [7]. In particular, we considered the CVEs examined by related work in the past 10 years and focused on those for which manual program inspection was realistic in the time available. In fact, rather than running known or self-crafted exploits to determine whether they would be (trivially) stopped, we decided to manually analyze the CVEs to determine if and under which conditions they would still be exploitable. This approach is time-consuming, but the benefit is that it allows us to determine not only if existing exploits still work, but also whether attackers could craft new Type-After-Type-aware exploits by taking into account the changes in memory layout.

For all the temporal heap CVEs (the first 7 entries) in Table 1, Type-After-Type's heap allocator and type analysis completely prevent attacks. Some programs implement a custom (e.g., slab-based) heap allocator. Type-After-Type supports system allocator functions (e.g., `malloc` and `free`) and their wrappers, but does not derive custom allocator semantics. To mitigate the vulnerabilities, the programmer must replace the use of custom allocators with direct use of the system allocator. This is typically configurable by means of a configuration flag before compilation. Alternatively, the programmer can specify allocations functions to our protection pass. For example, Firefox defines a custom C++ `new` operator that uses `moz_xmalloc` to perform allocations. Since these are used consistently, supporting them is straightforward.

Besides temporal memory attacks, Type-After-Type complicates any other exploits that rely on spatial type locality. For instance, it stops the type confusion exploit against Chrome of the 2013 Pwn2Own competition, as it relies on type-unsafe heap massaging. While one cannot rule out a Type-After-Type-aware exploit, this is challenging as the CVE only allows limited out-of-bounds read/write accesses past the vulnerable object. Hence, landing neighboring objects of an *attacker-controlled type* is crucial to (i) break ASLR (out-of-bounds read) and (ii) corrupt sensitive data (out-of-bounds write) for the exploit.

The final 3 entries in Table 1 describe stack CVEs. Type-After-Type's stack allocator and precise type analysis completely prevent attacks based on the first two CVEs. The third CVE is more interesting, as an attacker only needs to reuse (even type-safe) memory with non-zero values to lure the program into reading a stack-allocated string without a `'\0'` (uninitialized) terminator. This allows a Type-After-Type-aware attacker to craft a buffer overread primitive and leak information. Nonetheless, thanks to Type-After-Type, the attacker can again only access neighboring objects of the same type. This stops the exploit and frustrates more sophisticated exploitation attempts to leak "interesting" information. We note that our analysis here focuses on uninitialized reads for the stack, where, in contrast to the heap, such CVEs are more common than use-after-frees. Nonetheless, unlike existing practical heap-based [2] and stack-based protection mechanisms [22, 25, 27], Type-After-Type can also stop attacks based on stack use-after-frees such as CVE-2011-2013 and CVE-2014-3153.

8.2 Type detection

As discussed in Section 5.1, we use several approaches to detect the types of heap allocations. In this section, we determine to what extent our static type detection techniques complement each other.

We measured our static type detection results for the SPEC CPU2006 [18] benchmarks. Our system manages to detect the type for 86.9% of the `new` call sites and 56.8% of the other call sites. In the remaining cases, Type-After-Type falls back to run-time type detection (reducing performance, however marginally). Over all 6,793 non-`new` call sites together, cast-based detection yields 3,609 types and `sizeof`-based detection yields 1,311. Although this shows that cast-based detection is more effective overall, a number of benchmarks (milc, sjeng, libquantum, and h264ref) have the majority of their call sites resolved by the `sizeof` approach. This shows that the two approaches complement each other and the most effective one is subject to coding style.

To verify our type detection accuracy absent complete ground truth, we cross-check the consistency of the results between our two approaches. We found the typecast approach and the `sizeof` approach to yield slightly inconsistent results only for two benchmarks: povray (42 out of 897 call sites where both are available) and h264ref (9 out of 43). Given the highly consistent results, we conclude our type detection is reasonably accurate for the vast majority of the call sites.

One other point to note is that we do not consider `char*` to be a detected type. We cannot distinguish it from the untyped pointer `void*` because LLVM uses the same representation for both. The `char` type is also not commonly used with the `sizeof` operator because the C standard guarantees the type to be a single byte. Moreover, it is not desirable to pool `char*` allocations from different allocation sites because they often store untyped binary data in

practice. The fact that we cannot consider `char*` as a separate type does mean that our type accuracy could not reach 100% even with perfect type detection. A similar situation holds for allocations with the `new` allocator because the `new char[]` array allocators also yields an effectively typeless block of bytes.

8.3 Wrapper detection and inlining

Over all the SPEC CPU2006 [18] benchmarks, we found 802 possible wrapper functions. This number is considerably larger than the number of real wrapper functions one would expect, confirming our expectation that our wrapper detection yields many false positives As discussed in Section 5.2, such false positives do not harm security (in fact, they may even improve it), the only risk being that they may negatively impact performance.

The total number of allocation call sites increases from 6,104 to 8,243, an increase of 35%. This means that we inlined 2,139 calls to potential wrappers. While this seems like a relatively small increase, the impact is much larger for some specific benchmarks. In particular, the number of call sites becomes several times the original number for perlbench, gobmk, and povray. To verify our wrapper inlining accuracy absent a complete ground truth, we can check whether inlining a potential wrapper allow us to determine the type for an allocation site. Over all benchmarks, wrapper inlining increases the number of call sites with a type identified from 3,474 to 5,121, which suggests we do find a large number of true wrappers. This serves as a lower bound for the number of allocation call sites we managed to move out of wrappers because there may also be even more cases where either our static analysis is insufficient to determine the type after inlining or the type is `char*`. Inlining is especially effective for type detection for perlbench, gobmk, and povray. We conclude that our wrapper inlining is effective in reducing the impact of wrapper functions around memory allocations for these particular cases.

8.4 Memory overhead

Figure 2 shows our memory overhead on SPEC CPU2006 [18]. We compare our results against a baseline without instrumentation, using an unmodified version of the tcmalloc memory allocator [12]. We opted to use tcmalloc as a baseline rather than the less performance and memory-efficient system default allocator, given that ascribing the improvements resulting from the use of tcmalloc to our own system would give results that do not provide much information about the overhead incurred by type-safe memory reuse.

We measured the peak residual set size (RSS) using Linux' `time` utility to determine how much memory each benchmark requires. The geomean memory overhead for heap protection is just 17.3%, while stack protection incurs negligible overhead. Combined protection of both the stack and the heap incurs 17.4% memory overhead, just like heap protection by itself. As such, we conclude that overall memory overhead is reasonable given the strong protections offered by Type-After-Type.

8.5 Run-time overhead

To assess Type-After-Type's performance impact, we instrumented all C/C++ benchmarks of the SPEC CPU2006 [18] benchmarking

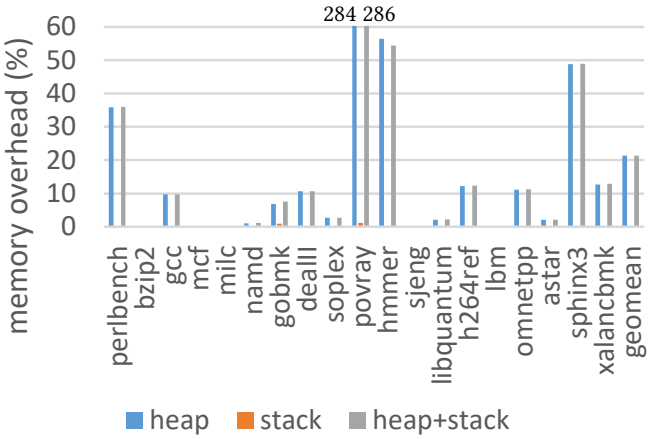

Figure 2: Memory overhead on SPEC CPU2006

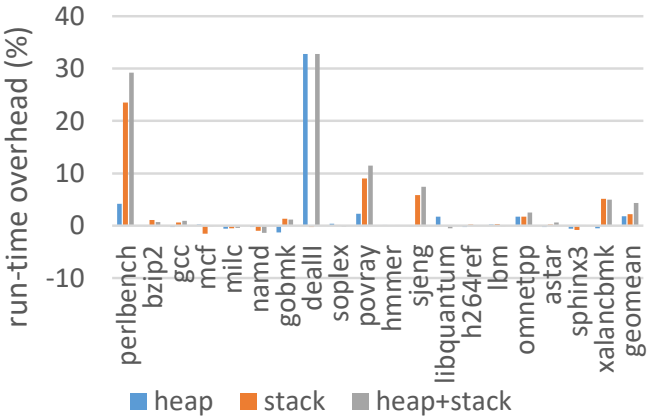

Figure 3: Performance overhead on SPEC CPU2006

suite using the default workload (reference). Again, we compare against a baseline using tcmalloc.

Figure 3 shows the results of our performance measurements. The geometric mean of the overhead of our solution compared to the baseline is 2.0% for heap protection, 2.0% for stack protection and 4.3% for both. We used our wrapper inlining pass to measure these results. Without wrapper inlining, the geometric mean of the overhead is 2.0% for just heap protection and 4.1% to protect both the heap and the stack. These numbers show that both heap and stack protection incur very low overhead even on memory-intensive benchmarks like SPEC's, reducing the performance cost to address temporal memory errors to a level where practical adoption is viable. It also shows that, despite our choice to to favor security over performance in wrapper detection, the performance impact of wrapper inlining is negligible.

Although our overall run-time overhead is very low, two benchmarks do incur substantial overhead. On perlbench, Type-After-Type incurs a total overhead of 24.2%, mostly due to stack protection. We found that one of the functions that executes much of the time in perlbench consists of a giant `switch` statement, in which some cases use array buffers that must be moved to the typed unsafe stacks. For each type of these buffers, Type-After-Type subtracts the buffer sizes from the typed stack pointer and stores the new

stack pointer as a local variable to allow access to the variables allocated on the unsafe stack. Unfortunately, the unusually large number of typed stacks used in the function causes the compiler to allocate these local variables on the safe stack rather than in CPU registers. Therefore, each buffer access requires an extra memory access to look up the typed stack pointer. This is not an issue for programs with a reasonably simple inner loop. Moreover, it could be solved with an improved register allocator assigning registers locally for the individual `switch` cases. The second case is dealII, where Type-After-Type incurs 35.7% run-time overhead, entirely due to heap protection. It spends an excessive amount of time performing an operation specific to tcmalloc, namely scavenging the per-thread free lists for entries to move to the central free list. As such, the particular sequence of allocations and deallocations in dealII causes pathological behavior in tcmalloc (amplified by typed heaps), which could be fixed by fine-tuning tcmalloc's thread cache scavenging settings. Moreover, in this particular case the operation could in principle be removed entirely because dealII is not multithreaded. Doing so eliminates almost all heap protection performance overhead. We conclude that, although Type-After-Type incurs non-trivial overhead in specific situations, these are pathological cases that in principle can be solved by fine-tuning the register or memory allocator to such scenarios.

To compare performance against state-of-the-art (heap-based) solutions, Cling [2] in particular, we also compare our heap protection performance against the system default allocator rather than our regular baseline. Cling also compares against the system default allocator, which means these measurements cannot distinguish the overhead from type-safe memory reuse from the speedup introduced by using a more efficient memory allocator. Our geometric mean over all benchmarks is -6.0%, so the benefit from using an efficient allocator (tcmalloc) is larger than the overhead introduced by Type-After-Type, resulting in an overall speedup. Our geometric mean over the SPEC CPU2006 benchmarks evaluated by Cling is -7.1% on the heap and -5.8% for stack+heap, while Cling achieves -4.3% for just the heap. This means that Type-After-Type compares well to the state of the art, and achieves a greater speedup compared to the system default despite the complete (heap+stack) protection.

8.6 Firefox case study

In addition to our SPEC CPU2006 experiments, we applied our instrumentation to the Mozilla Firefox browser [41] to measure the impact on a memory-demanding application that is heavily object-oriented and a major target for attacks based on temporal memory errors. We ran the Sunspider 1.0.2 [16], Octane 2.0 [15], and Dromaeo [11] Javascript and DOM benchmarks to test its performance. These are demanding CPU-intensive workloads that provide a good indication of the worst-case performance impact.

Table 2 shows our results for heap and stack protection on Firefox 47.0. Unfortunately, we could only apply stack protection to Firefox' main binary and not to its libraries, because our SafeStack baseline does not support shared libraries and Firefox support for building with static libraries is currently broken. We do not inline wrappers for Firefox because crashes, but fortunately overhead from inlining wrappers is negligible (see Section 8.5) so we expect this has little impact on the results. Type-After-Type can protect Firefox with very low overhead even for these compute-intensive workloads. Moverover, in most practical settings these overheads will be hidden by I/O latencies. As such, we believe that our system is well within the range of acceptable overheads for production use. However, we caution the reader that the work performed in Firefox' shared libraries is not neglible, and that extending the stack protection to these libraries is likely to increase overhead. Nonetheless, we do not expect the overheads to deviate significantly from those of an improved SafeStack, a now-standard protection mechanism in the clang/llvm production compiler.

benchmark	overhead		
	heap	stack	stack+heap
SunSpider	2.2%	1.9%	2.9%
Octane 2.0	-0.7%	0.9%	0.9%
Dromaeo JS	4.0%	-1.6%	5.2%
Dromaeo DOM	0.6%	-1.4%	0.9%

Table 2: Overhead% Firefox (stack only main binary).

In some cases, we measured negative overheads even though the instrumentation we add should not make the program faster. These results are consistent between measurements and we believe this to be caused by the performance variations due to small differences in memory layout [29] or possibly type-based pooling improving locality and thus cache performance in certain scenarios.

Safestack does not support shared libraries because the linker does not merge the thread local variable used to store the unsafe stack pointer between the main binary and the shared libraries, resulting in each having their own local unsafe stack pointer, which is currently not initialized by the shared libraries. This results in a crash the first time a variable is stored on the unsafe stack.

It should be noted that the limitation that we cannot use our stack protection for shared libraries (inherited by SafeStack) is not fundamental. The best solution would be to modify the linker to make it merge thread local variables in shared libraries with existing thread local variables with the same name. Unfortunately, however, modifying the linker is not practical because the resulting binaries would not work on existing systems. However, it is also possible to work around this issue. One could add code in the shared libraries to make them initialize their own stack pointers for each existing thread when the library is loaded. In addition, one would need to keep track of a list of libraries in the main binary to initialize stack pointers for all shared libraries whenever a new thread is created. In addition, this would allow one to create a callback for stack unwinding that unwinds all stacks from all libraries rather than just the one in the current module. This solution would allow Safestack to protect shared libraries and would be equally applicable to our solution, which uses multiple unsafe stacks.

9 LIMITATIONS

Although our system can mitigate a wide range of temporal memory safety errors at a very low performance cost and is compatible with a wide range of programs, our design has a number of limitations which we will discuss in this section.

First, there is a number of attacks against temporal memory errors that our system cannot prevent. One instance is the situation

where the memory error can be exploited without violating type safety. For example, in case of use-after-free vulnerabilities, this means that the attacker can use a dangling pointer to manipulate a live object of the same type. Given that pointers are generally not user-controllable, this means that the attacker will generally not be able to manipulate pointers, denying the attacker access to a powerful arbitrary read and/or write primitive and making them unable to leak pointers to defeat address space layout randomization (ASLR). Instead, the attacker has to resort to more subtle attacks, such as changing a field after its value has been verified to be safe (Time-of-check-to-time-of-use). While such attacks will be possible in some cases, they do depend on being able to perform very particular sequences of operations, dramatically shrinking the attack surface of modern exploits. A second instance where we cannot prevent attacks is the case where the programmers themselves introduce deliberate type unsafety. In particular, this can happen due to the use of implicit or explicit C/C++ unions. As the previous case, however, an attack against such type unsafety depends on a very specific sequence of operations. Normally, the attacker can generally perform arbitrary type-unsafe accesses via memory massaging, while with Type-After-Type (type-unsafe) massaging is impossible and only unsafe type conversions introduced by the programmer can be abused.

A second limitation is the inability to handle memory allocated directly from the OS through system calls such as `brk` and `mmap`. However, the `brk` call is rarely used directly by applications due to its inability to deallocate memory out-of-order. The other cases are merely implementation limitations and they could be handled by intercepting calls such as `mmap` and similar.

A third limitation is the fact that, if a call site can allocate multiple types, we cannot prevent type-unsafe reuse. In this case, when we detect multiple types can be allocated we use a call site hash rather than a type hash to keep the typed memory pools free from other types, but memory can be reused for the same call site. In many cases, our wrapper detection and inlining prevents this issue by duplicating the call site for each type allocated, but there are some cases that cannot be handled by inlining. An example would be the case where the function selects a type in an `if` or `switch` statement that cannot be eliminated by optimizations after inlining. This limitation is in common with Cling [2], which cannot distinguish types on identical allocation call stacks. However, unlike Cling, in our case this is a limitation that could in principle be addressed. If we detect such a case, we could duplicate basic blocks to create different allocation call sites for each type-specific path in the control flow graph leading to the original call site. This would effectively amount to inlining at a finer granularity.

A fourth limitation is that we do not handle (less common) indirect calls to allocation functions or wrappers. This means that in the former case the allocated memory will be typeless and shared with other uninstrumented allocations (e.g., in uninstrumented libraries) while in the latter case a call site hash will be generated rather than a type hash, possibly resulting in the issues described earlier. This issue could be solved through additional static analysis to determine which function pointers may be passed to which indirect call sites, for example using type-based callgraph analysis [25]. One could then add code to check whether the predicted callee is correct and, if so, perform a direct call instead. This direct call can then be used in the analysis performed to detect and inline wrappers and can be identified as an allocation call site in our type detection pass.

A fifth limitation is the fact that we cannot determine types if the protected program uses an arbitrary (non-wrapper-like) custom memory allocator. Fortunately, such approaches are often unnecessary with tcmalloc, which is often considerably faster than the system default allocator in handling cases where such constructs are often used. Therefore, disabling custom memory allocators—which generally requires only small configuration changes—is a simple solution to this issue.

Finally, our solution requires source code as it requires compiler-managed type information. While Cling [2] works around this issue by using call stack hashes as proxies for heap object types, this is not a viable solution for the frequent stack allocations that we support. As such, we believe the source code requirement is inherent to a complete temporal type safety solution.

10 RELATED WORK

With the rise of use-after-free and similar exploits in recent years comes increasing interest in temporal memory safety in the research community. While it is impossible to exhaustively list all related publications, we distinguish between defenses that make it hard for the attacker to (re-)allocate useful areas of memory, defenses that remove dangling pointers, defenses that explicitly check the validity of objects on dereferences, and defenses that explicitly initialize memory prior to reuse.

Secure memory allocators. Google Chrome [14] uses‘ the PartitionAlloc allocation library, which, like Type-After-Type, partitions heap allocations by class. However, the class has to be manually specified by the developer for each allocation. In contrast, Type-After-Type only requires the build system to specify the correct compiler (flags), which makes adoption considerably easier.

Like Type-After-Type, Cling [2] offers type-safe memory reuse, but with important differences. First, it does not at all support the stack, leaving software vulnerable to widespread temporal stack vulnerabilities (and this is also the case for more recent variants focusing only on heap C++ objects [35]). Second, it is more limited in the detection of types and wrappers. In particular, it infers information about allocated objects and wrapper functions based on the run-time callstack, which is less efficient and less effective than Type-After-Type's compile-time analysis. Moreover, Cling can only detect wrappers at runtime when it observes a single allocation site that requests multiple allocation sizes, which is neither always reliable, nor immediate, introducing a window of vulnerability. It also limits the number of levels of wrappers supported. In contrast, Type-After-Type does not rely on sizes for wrapper detection at all. Finally, Cling completely replaces the existing allocator with a new implementation, while Type-After-Type's prototype builds on the high-performance tcmalloc allocator widely used in production.

An early secure allocator is DieHard [4], which uses randomization to provide probabilistic temporal safety. After DieHard, a variety of randomization-based secure allocators have emerged. Most of them focus on the heap, but some exclusively target the stack [6]. Its most direct successor, DieHarder [32] built has fairly high overhead (up to 2x and 20% geomean on SPECint). Moreover,

like all the other randomized allocators and also more recent approaches such as ASan [38], this approach can be bypassed by massaging memory to force type-unsafe reuse.

SafeCode [9] offers a compiler-based system for type-safe memory reuse on the heap which depends on data structure analysis (DSA). It uses a custom pool allocator that frees a pool after the last referencing pointer is dropped. The approach relies on the assumption that static points-to analysis is capable of perfectly determining control flow (including indirect calls) and data flow. Although this may hold for some embedded applications (which are the target of SafeCode), this is unfortunately not the case for complex programs, as DSA has been shown to "lose field sensitivity on up to 70.5% of tracked struct objects" [10]. Our analysis, while simpler, has better worst-case performance and is type safe as long as wrappers are inlined correctly. Moreover, our solution performs type-safe allocations on both the stack and the heap.

Dangling pointers. Rather than securing the allocation process, some defenses invalidate all pointers to a freed object immediately [8, 24, 42, 45]. Unfortunately, this strategy incurs non-trivial overhead (due to pointer tracking or page protection changes), especially in the presence of multi-threading. Compared to DangSan [42] and Oscar [8], the most advanced solutions to date, the overhead of Type-After-Type is an order of magnitude lower. Other approaches [3, 30, 43, 44] instead check the validity of pointers on a dereference, but this results in even higher overhead (e.g., 48% on average for CETS [30]).

Uninitialized reads. Finally, defenses may ensure that even if predictable memory reuse is possible and dangling pointers remain, the reused memory areas do not contain anything interesting—by re-initializing the pages either at deallocation time [32], or just prior to their use by untrusted code [25, 27]. Doing so at deallocation time is expensive, and recent approaches instead opt to zero out memory selectively and "just-in-time". For instance, UniSan [25] protects memory disclosures from the kernel by zeroing out memory only if such bytes may cross into userspace without initialization. Likewise, SafeInit [27] zeros out memory at allocation time, only if there is a chance that the program may use it uninitialized.

Compared to Type-After-Type, SafeInit (the state-of-the-art uninitialized read protection for generic programs) cannot address use-after-free vulnerabilities. In addition, SafeInit's optimization passes have trouble scaling to large buffers on the stack involved in complex data flows, resulting in non-trivial overhead when unnecessarily zeroing unused stack buffers [27]. Type-After-Type moves such buffers to a per-type stack, eliminating unnecessary zero initialization and improving performance in such problematic (but common) cases.

Garbage collection. One approach to prevent dangling pointers is to automatically free objects whenever this is safe to do. Garbage collection is widely used in languages designed to be safe, such as Java [34] and Go [1]. In all cases, however, safe languages semantically differ too much from traditional unsafe languages such as C and C++ to allow programs to be converted automatically, resulting in a considerable porting effort [36]. The same holds for new dialects of C or C++, such as Cyclone [19] and CCured [31], which change memory management and introduce new primitives to make the C language safer. However, just like with new safe languages, considerable porting effort is needed so existing programs are rarely converted in practice.

An alternative is to keep the programming language as-is, but add a garbage collection library such as the Boehm garbage collector [5]. This approach has already been adopted in some mainstream browsers, such as MemGC in Microsoft's Edge browser [40] and Oilpan in Google's Chrome browser [13]. Unfortunately, however, C and C++ do not keep track of the necessary type information and many common C idioms as well as some compiler optimizations make it impossible for the garbage collector to reliably distinguish pointers and integers, which is necessary to be able to correctly find references to objects. As such, in practice non-trivial programs must either be developed from scratch to work with a conservative collector or existing programs must be modified to provide annotations that give the garbage collector the information that it needs. Conservative garbage collectors also introduce side channels that may be abused by attackers to bypass address space layout randomization (ASLR) [17]. As such, garbage collectors offer neither the convenience nor the security provided by Type-After-Type.

Finally, there is the option of keeping explicit memory management but delaying free operations until a fixed point in the program is reached to prevent memory reuse within the same scope as the free operation. This delayed free approach is currently used in Microsoft's Internet Explorer [39]. Although this is a simple solution that raises the bar for attackers, memory is reused eventually so that references used outside the scope where they were created are still a threat. Moreover, this approach allows memory usage to increase without bound. This may be useful for attackers to perform heap spraying or craft allocation oracles [33]. As such, Type-After-Type offers better security than delay free approaches.

11 CONCLUSION

Temporal memory errors are a major threat to software security. Existing solutions to address this problem either incur excessive overhead for practical use, can be subverted via memory massaging, or do not provide complete protection for stack and heap. We presented Type-After-Type, a novel design to address temporal memory errors through practical and complete type-safe memory reuse. We used compiler instrumentation to create separate stacks for each type and to annotate heap allocations with a type to support per-type heap pools. We showed that Type-After-Type achieves complete temporal type safety at just 4.3% overhead, matching the golden 5% standard for real-world production deployment.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct), by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by Cisco Systems, Inc. through grant #1138109, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO 639.021.753 VENI "PantaRhei". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] [n. d.]. The Go Programming Language. https://www.golang.org.
[2] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers.. In *USENIX Security*.
[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. *SIGPLAN Not.* 29, 6 (June 1994), 290–301. https://doi.org/10.1145/773473.178446
[4] Emery D Berger and Benjamin G Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*.
[5] Hans Boehm. [n. d.]. A garbage collector for C and C++. http://www.hboehm.info/gc/.
[6] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries.. In *NDSS*.
[7] The MITRE Corporation. [n. d.]. Common Vulnerabilities and Exposures. http://cve.mitre.org/.
[8] Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security*.
[9] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2005. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 1 (2005), 73–111.
[10] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS*.
[11] The Mozilla Foundation. [n. d.]. DROMAEO, JavaScript Performance Testing. https://www.webkit.org/perf/sunspider/sunspider.html.
[12] S. Ghemawat and P. Menage. [n. d.]. Thread-caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
[13] Google. [n. d.]. Blink GC API reference. https://chromium.googlesource.com/chromium/src/+/master/third_party/WebKit/Source/platform/heap/BlinkGCAPIReference.md.
[14] Google. [n. d.]. Chrome. https://www.google.com/chrome/.
[15] Google. [n. d.]. Octane Benchmark. https://code.google.com/p/octane-benchmark.
[16] Google. [n. d.]. SunSpider Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html.
[17] Abdul-Aziz Hariri, Brian Gorenc, and Simon Zuckerbraun. 2015. Abusing Silent Mitigations: Understanding weaknesses within Internet Explorer's Isolated Heap and MemoryProtection. In *Black Hat USA*.
[18] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
[19] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC*.
[20] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: Buffer overflow checks without the checks. In *EuroSys*.
[21] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBounds: Memory Safety for Shielded Execution. In *EuroSys*.
[22] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *OSDI*.
[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.
[24] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*.
[25] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *CCS*.
[26] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *NDSS*.
[27] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *NDSS*.
[28] Matt Miller. 2017. Uninitialized use by the containing memory region type (stack, heap, other). http://pic.twitter.com/Buj5xJaPD0.
[29] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices* 44, 3 (2009), 265–276.
[30] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: Compiler-enforced Temporal Safety for C. In *ISMM*.
[31] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892
[32] Gene Novark and Emery D Berger. 2010. DieHarder: Securing the Heap. In *CCS*.
[33] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security*.
[34] Oracle. [n. d.]. Java. https://www.java.com/.
[35] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering class hierarchies in C++ programs. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*.
[36] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C *(ISMM)*.
[37] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *ACSAC*.
[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.
[39] Jack Tang. 2014. Mitigating UAF Exploits with Delay Free for Internet Explorer. http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/.
[40] Microsoft Edge Team. 2015. Microsoft Edge: Building a safer browser. https://blogs.windows.com/msedgedev/2015/05/11/microsoft-edge-building-a-safer-browser/.
[41] The Mozilla Foundation. [n. d.]. Mozilla Firefox. https://www.mozilla.org/firefox.
[42] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *EuroSys*.
[43] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 117–126. https://doi.org/10.1145/1041685.1029913
[44] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. ACM, New York, NY, USA, 307–316. https://doi.org/10.1145/940071.940113
[45] Yves Younan. 2015. FreeSentry: Protecting against Use-after-free Vulnerabilities due to Dangling Pointers. In *NDSS*.