SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks

Dongseok Jang
d1jang@cs.ucsd.edu
Computer Science and Engineering
University of California, San Diego

Zachary Tatlock
ztatlock@cs.washington.edu
Computer Science and Engineering
University of Washington

Sorin Lerner
lerner@cs.ucsd.edu
Computer Science and Engineering
University of California, San Diego

***Abstract*—Several defenses have increased the cost of traditional, low-level attacks that corrupt control data, e.g. return addresses saved on the stack, to compromise program execution. In response, creative adversaries have begun circumventing these defenses by exploiting programming errors to manipulate pointers to virtual tables, or *vtables*, of C++ objects. These attacks can hijack program control flow whenever a virtual method of a corrupted object is called, potentially allowing the attacker to gain complete control of the underlying system. In this paper we present SafeDispatch, a novel defense to prevent such *vtable hijacking* by statically analyzing C++ programs and inserting sufficient runtime checks to ensure that control flow at virtual method call sites cannot be arbitrarily influenced by an attacker. We implemented SafeDispatch as a Clang++/LLVM extension, used our enhanced compiler to build a vtable-safe version of the Google Chromium browser, and measured the performance overhead of our approach on popular browser benchmark suites. By carefully crafting a handful of optimizations, we were able to reduce average runtime overhead to just 2.1%.**

I. Introduction

Applications like web browsers and office productivity suites are increasingly trusted to store and manipulate highly sensitive data in domains ranging from medical record management to banking. Such systems demand both performance and abstraction, making a low-level, object-oriented language like C++ the tool of choice for their implementation. Unfortunately, this focus on performance has all too often taken precedence over critical security concerns. Malicious attacks frequently exploit the low-level programming errors that plague these systems, allowing an adversary to corrupt *control data*, pointers to code which the program later jumps to. By compromising control data, attackers are able to hijack program execution, in the worst case leading to arbitrary code execution.

Buffer overflows are one of the most familiar techniques for corrupting control data: by overwriting the return address in a function's activation record on the stack, the attacker can specify which instruction the CPU will jump to when the function returns, thus hijacking the program's execution. The security community has responded to such attacks with numerous defenses, including stack canaries [1], data execution prevention [2], and custom allocators to protect the heap [3]. These successful defenses have increased the cost of mounting traditional attacks, forcing adversaries to adopt increasingly sophisticated approaches.

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/doi-info-to-be-provided-later

Instead of overwriting return addresses saved on the stack, several recent, high profile attacks have shifted their focus to corrupting another class of control data: heap-based pointers to virtual tables, or *vtables*. A C++ class's vtable contains function pointers to the implementations for each of its methods. All major C++ compilers, including GCC, Visual C++, and LLVM, use vtables to implement dynamic dispatch: whenever an object invokes a virtual method, the vtable for that object's class is consulted to determine which function should be called. This layer of indirection enables polymorphism in C++ by allowing a subclass to invoke its own version of a method, overriding its parent class.

For performance, the first word of a C++ object with virtual methods is a pointer to its class's vtable. Unfortunately, this efficiency comes at a price: memory safety violations can nullify an important invariant: *the vtable pointer stored in an object of type τ always points to the vtable of τ or one of its subclasses*. If an attacker can corrupt an object's vtable pointer to instead point to a counterfeit vtable, then they can hijack program control flow whenever that object calls one of its virtual methods, potentially executing malicious shellcode [4]. In this paper, we call such attacks *vtable hijacking* and describe an efficient technique to prevent them.

Security researchers previously demonstrated one of the *many* ways an attacker can hijack vtables: by exploiting use-after-free errors. In this particular attack method, an adversary first identifies a dangling pointer, a reference to an object that has been freed. The attacker then tricks the program into allocating both: (1) a counterfeit vtable and (2) a pointer to this counterfeit vtable *at the start of the memory where the freed object was stored*. Finally, the attacker manipulates the program to invoke a virtual method via the dangling pointer. Because the attacker has overwritten the vtable pointer in the freed object, this method call will jump to an address of the attacker's choosing, as specified by their counterfeit vtable. Exploiting such use-after-free errors is just one way to launch vtable hijakcing attacks, others include traditional buffer overflows on the stack or the heap [4] and type confusion [5], [6] attacks. Unfortunately, such vtable hijacking attacks are no longer merely a hypothetical threat [7], [8].

We increasingly observe robust vtable hijacking attacks in the wild, often leading to the execution of malicious shellcode. Such attacks have recently been shown practical in complex applications, including major web browsers: in recent Pwn2Own competitions, vtable hijacking enabled multiple arbitrary code execution attacks in Google Chrome [9], Internet Explorer [10], and Mozilla Firefox [11]. In fact, abusing dynamic dispatch in C++ was the major security weakness in all these browsers. In a recent Google Chrome exploit, Pinkie Pie employed a vtable hijacking attack to construct a Zero-day vulnerability to escape the tab sandbox and execute arbitrary code [12]. As a result of such attacks, researchers have recently singled out vtable hijacking as one of the most straightforward attack vectors exploiting heap vulnerabilities, as an attacker can often construct inputs to influence when a program allocates and frees objects.

Unfortunately, existing defenses that could prevent vtable hijacking are either incomplete or do not specifically take advantage of the C++ type system to provide the best possible performance. Techniques like reference counting can help mitigate vtable hijacking attacks that exploit dangling pointers, e.g. by preventing dangling pointers from being used for invoking methods. Unfortunately, there are many other ways to mount vtable hijacking attacks that do not require a dangling pointer. Other techniques like control flow integrity [13], [14], [15], [16], [17] can secure all indirect jumps to prevent many kinds of control flow hijacking attacks, including vtable hijacking. However, these techniques do not take advantage of the C++ type system for the specific task of securing virtual method calls, and therefore none of these techniques treat C++ virtual method calls *both* precisely and efficiently.

In this paper, we address the growing threat of vtable hijacking with SAFEDISPATCH, an enhanced C++ compiler that prevents such attacks. SAFEDISPATCH first performs a static class hierarchy analysis (CHA) to determine, for each class c in the program, the set of valid method implementations that may be invoked by an object of static type c. SAFEDISPATCH uses this information to instrument the program with dynamic checks, ensuring that, at runtime, all method calls invoke a valid method implementation according to C++ dynamic dispatch rules. By carefully optimizing these checks, we were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5% in the first vtable-safe version of the Google Chromium browser which we built with the SAFEDISPATCH compiler.

To summarize, this paper makes the following contributions:

- We develop SAFEDISPATCH, a comprehensive defense against vtable hijacking attacks. We detail the static analysis and compilation techniques to efficiently ensure control flow integrity through virtual method calls.
- We detail the implementation of SAFEDISPATCH as an enhanced C++ compiler and discuss several security and performance trade offs that influenced our design.
- We applied SAFEDISPATCH to the entire Google Chromium web browser code base to evaluate the effectiveness and efficiency of our approach. By developing a handful of carefully crafted optimizations, we were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5%.

```
// for displaying content on screen
class Window: {
  public: virtual void display(string s) { ... }
};

// specialized for small screens on mobile devices
class MobileWin: public Window {
  public: virtual void display(string s) { ... }
};

Window* w = flag ? new Window() : new MobileWin();
w->display("Hello");      // invoke virtual method
delete w;                 // free w, now dangling
          _ _ _ _ _ _ _ _ _ _ _

// behavior of code generated for w->display("Hello")
typedef void*   method;    // method is func ptr of any type
typedef method* vtable;    // vtable is array of methods
vtable t = *((vtable *)w); // 1. vtable @ 1st word of object
method m = t[0];           // 2. lookup by display's id, 0
m(w, "Hello");             // 3. make virtual call
```

Fig. 1. **C++ Dynamic Dispatch.** Consider the simple `Window` class above for displaying a string on the screen. C++ compilers translate each virtual method call into lower level code that performs three steps: (1) dereference the first word of the calling object to retrieve its class's vtable, (2) index into the vtable by the method's position in the class to retrieve the appropriate function pointer, and (3) call the retrieved function pointer, passing the calling object as the first argument, followed by any additional arguments. If an attacker corrupts an object's vtable pointer to point to a counterfeit vtable, possibly by exploiting a dangling pointer, then they can cause steps (1) and (2) to lookup malicious code and step (3) to execute it.

In the next section we provide additional background on C++ dynamic dispatch and vtable hijacking and then overview how SAFEDISPATCH prevents such attacks. Section III follows, where we detail the SAFEDISPATCH compiler, key optimizations we developed to minimize overhead, and some of the different security and performance tradeoffs we considered. Next, in Section VI, we evaluate our SAFEDISPATCH implementation along several dimensions, including performance overhead, while in Section VII we discuss the security implications of our approach. In Section VIII we survey existing defenses, discussing their effectiveness at mitigating vtable hijacking in complex, high performance systems and comparing them with SAFEDISPATCH. Finally, in Section IX we consider future directions and conclude.

II. SAFEDISPATCH OVERVIEW

In this section we provide additional background on dynamic dispatch in C++, illustrate vtable hijacking with a detailed example, and provide a high level description of how SAFEDISPATCH prevents such attacks.

A. Dynamic Dispatch in C++

Before detailing an example vtable hijacking attack, we briefly review how dynamic dispatch invokes object methods in C++. Consider the code in the upper part of Figure 1, which declares two classes: a `Window` class with one virtual method named `display` for displaying a string on the screen and a `MobileWin` subclass of `Window` which overrides `display` to provide an implementation specialized for smaller screens.

C++ dynamic dispatch rules dictate that when an object calls a virtual method, the actual implementation invoked depends on the *runtime type* of the calling object. This layer of indirection allows subclasses to override their parent class's implementation of methods and is one of the key mechanisms for polymorphism in C++. For example, in the code snippet from Figure 1, the call `w->display("Hello")` will either invoke `Window::display` or `MobileWin::display`, depending on what `w` refers to at run-time, which in turn is determined by the `flag` variable.

Of the many implementation strategies for dynamic dispatch, *Virtual Method Tables*, or vtables are the most common. Prevalent C++ compilers, including GCC, Visual C++, and Clang++, all use vtables due to their efficiency. To implement vtables, the compiler assigns each virtual method in a class an identifier, which for simplicity we assume is done by numbering virtual methods sequentially. A vtable for class C is then an array t such that $t[i]$ is the implementation of method i for class C. At compile time, the compiler constructs a vtable for each class, and inserts code in the constructor of each class to initialize the first word of the constructed object with a pointer to the vtable for that class.

To implement a virtual method call the compiler generates code that performs three steps: (1) load the vtable pointer, located at position 0 in the calling object, (2) lookup index i in the vtable, where i is the index of the method being called (3) call the method implementation found at index i in the vtable. The lower part of Figure 1 uses C++ notation to illustrate the behavior of code generated for `w->display("Hi")`, assuming that `display` is given index 0 by the compiler. Note that if `w` points to a `Window` object, then the vtable will contain `Window::display` at location 0, whereas if `w` points to a `MobileWin` object, then the vtable will contain `MobileWin::display` at location 0.

Because vtables are used in determining control flow, if an attacker can illegally manipulate an object's vtable pointer, they can hijack program execution whenever that object invokes a virtual method. Since objects are ubiquitous in C++ programs, such control data is abundant, making vtable hijacking an attractive target for adversaries seeking to exploit low-level programming errors. We next illustrate how an attacker may mount such attacks.

B. vtable Hijacking

Having reviewed C++ dynamic dispatch, we now illustrate an example of vtable hijacking using the code in Figure 2. This code mimics the structure of a browser kernel in the style of OP [18] or Google Chrome [19], [20]. In these browsers, tabs run as separate, strictly sandboxed, processes whose only capability is communicating with the browser kernel process. To perform privileged operations, e.g. rendering to the screen or initiating a network connection, a tab process must send requests to the browser kernel process which enforces access control for privileged operations. This architecture provides strong security properties: even fully compromising a tab does not immediately grant an attacker the ability to run arbitrary code since the tab sandbox prevents an exploited tab from performing any privileged operations. Of course, if the browser kernel contains an exploitable bug, the attacker may take full control of the underlying system.

```
class Shell {
  public: virtual string run(string cmd) { ... }
};

// for displaying content on screen
class Window: {
  public: virtual void display(string s) { ... }
};

// specialized for small screens on mobile devices
class MobileWin: public Window {
  public: virtual void display(string s) { ... }
};

void tab_request_handler_loop(void) {
  Shell*  sh  = NULL;
  Window* win = SMALL_SCREEN ? new MobileWin() : new Window();

  while (TRUE) {
    TabRequest r = recv_tab_request();
    switch (r.kind) {
      case GET_DATE:
        if (sh == NULL)
          sh = new Shell();
        // run shell with safe, const string
        string d = sh->run("date");
        send_tab_response(r.originating_tab, d);
        break;
      case DISPLAY_ALERT:
        win->display(r.msg);
        // equivalently:
        //   vtable t = *((vtable *)win);
        //   method m = t[0];
        //   m(win, r.msg)
        //
        // If the object that win points to was accidentally
        // deleted, and a Shell object was allocated in its
        // place, then the above call invokes method 0 of
        // Shell via the dangling win ptr, namely "run" with
        // a tab-controlled arg!
        break;
      case GET_HTML:
        ...
        // BUG: accidental delete, win ptr now dangling
        delete win;
        ...
        break;
    }
  }
}
            — — — — — — — — — — — —

// attack request sequence to run arbitrary shell command
GET_HTML, GET_DATE, DISPLAY_ALERT
```

Fig. 2. **Example vtable Hijacking.** The above code sketches the core of a browser kernel in the style of Google Chrome: tabs run as separate, strictly sandboxed processes and send requests to the kernel to perform privileged operations like running shell commands or accessing the network. The main loop above illustrates how such a browser kernel responds to unprivileged tab requests. Due to a use-after-free error, an attacker can craft a sequence of requests causing the above code to run arbitrary shell commands.

The attack we demonstrate here assumes an adversary has already compromised a tab process which they now use to mount an attack against the highly privileged browser kernel. Although the code in this example is greatly simplified, a similar attack was central to Pinkie Pie's 2012 Zero-day exploit against Google Chrome [12]. Furthermore, while this example shows how vtable hijacking can be used to compromise a browser kernel, the approach generalizes to mounting attacks against many kinds of software, allowing an adversary to hijack program control flow, and thus potentially execute malicious shellcode.

The core of Figure 2 depicts a loop inside the browser kernel to handle requests from unprivileged tab processes.

For this simplified example, we consider three handlers which together enable a vtable hijacking attack that will allow an adversary to execute an arbitrary shell command.

The handler for GET_DATE uses a Shell object to execute a shell command which retrieves the system's date information, and then sends the result back to the requesting tab. Note that the parameter passed to Shell::run is a safe, constant string.

The handler for DISPLAY_ALERT renders a tab-provided string to the screen using a Window object. According to the C++ type system, at runtime this object will be an instance of Window or any of its subclass. In this case, there are two possibilities, either the Window class or the MobileWin class, which is specialized to render on smaller screens, and is used depending on the setting in the SMALL_SCREEN variable flag.

These two handlers alone do not contain an exploitable bug. However, we now introduce a third handler for GET_HTML requests which, somewhere in the process of fetching HTML for a tab-provided URL, inadvertently deletes the Window object pointed to by win, leaving the win pointer dangling.

The attack now consists of the adversary controlled tab sending three requests: GET_HTML, GET_DATE, and DISPLAY_ALERT. First, when kernel processes the GET_HTML request, the win object is accidently deleted. Second, when the kernel processes the GET_DATE request, a new Shell object is allocated. The memory allocator may place this object *at the same memory location just freed by the previous handler*, leaving the dangling win pointer to refer to this newly allocated Shell object. Third, when the kernel processes the DISPLAY_ALERT request, the method call win->display(r.msg) dereferences the first word of win to get a vtable and calls the first function contained in that vtable. However, since win now points to a Shell object, its vtable pointer refers to Shell's vtable whose first element is the run method. Therefore, win->display(r.msg) actually calls Shell::run with r.msg as a parameter, a value provided by the attacker controlled tab. Thus, by sending these three requests in order, the compromised tab has tricked the kernel into running an arbitrary shell command, completely violating the kernel's security guarantee: the browser kernel's prime directive is to ensure all privileged operations are appropriately guarded, even in the face of a fully comprised tab processes.

This example illustrates just one of the *many* ways an attacker may mount a vtable hijacking attack. In addition to exploiting use-after-free errors, traditional buffer overflows (on the stack or heap), type confusion attacks, and vtable escape vulnerabilities are some of the techniques an attacker can employ to corrupt an object's vtable pointer and hijack program execution. We next sketch how SAFEDISPATCH prevents the attack shown in this example and consider the general case in subsequent sections.

C. SAFEDISPATCH *vtable Protection*

The attack illustrated in Figure 2 compromises control flow through the win->display(r.msg) method call to trick the program into invoking Shell::run(r.msg) instead.

```
// SAFEDISPATCH protection for win->display(r.msg)
vtable t = *((vtable *)win);    // load vtable
method m = t[0];                // lookup method
if(m == Window::display ||
   m == MobileWin::display)     // check ensures m valid
  m(win, r.msg);
else                            // otherwise, signal error
  error("bogus method implementation!");
```

Fig. 3. **SAFEDISPATCH Protection.** The SAFEDISPATCH compiler inserts checks at each method call site, analogous to those shown in **bold** above, to ensure that a method looked up from an object's vtable is valid given the object's static type, i.e. that it is a method of the object's class or one of its subclasses. Since our Window class has one subclass which overrides display, there are two valid methods in this case, Window::display and MobileWin::display. This check ensures that control flow through method calls satisfies the C++ type system, effectively preventing the attacker from executing arbitrary code. We detail our general approach in Section III.

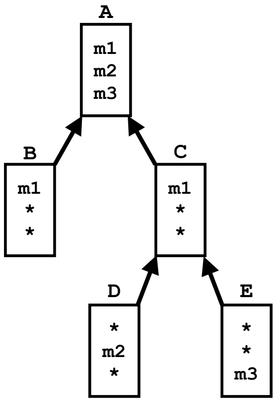
To prevent such attacks, SAFEDISPATCH inserts code to check the integrity of control-flow transfers for virtual method calls. In particular, at each virtual method call site, SAFEDISPATCH inserts checks to ensure that the code being invoked is a valid implementation of the called method according the static type of the object being called. For example, Figure 3 sketches the code that SAFEDISPATCH generates to protect the call win->display(r.msg). The additional checking code, shown in bold, guarantees that the method being called is either Window::display or MobileWin::display, which SAFEDISPATCH knows are the only two valid possibilities given the static type of win. This checking code not only prevents the previously described attack, but also adds only minimal overhead compared to the existing dynamic dispatch code.

So far, we have shown how SAFEDISPATCH prevents an attack on a simple example. In the remainder of the paper we explain how SAFEDISPATCH works in the general case, and present experimental results demonstrating that the overhead on complex, industrial scale applications is relatively low.

III. THE SAFEDISPATCH COMPILER

At their core, vtable hijacking attacks cause a virtual method call to jump into code which is not a valid implementation of that method. SAFEDISPATCH defends against all such attacks by instrumenting programs to ensure that, at every virtual method call site, the function pointer retrieved from the object's vtable *at runtime* is a valid implementation of the method being called (according to C++ dynamic dispatch rules), even if an attacker has managed to corrupt memory by exploiting a bug in the program.

In this section we describe our implementation of SAFEDISPATCH as an enahnced C++ compiler, built on top of the Clang++/LLVM compiler infrastructure [21]. SAFEDISPATCH extends this infrastructure with three major passes to insert checks which protect an application from vtable hijacking: (1) a variant of static Class Hierarchy Analysis [22] (CHA) which allows us to determine, *at compile time*, all the valid method implementations that may be invoked by an object of a particular static type at a given method call site, (2) a pass which uses the results from CHA to insert runtime checks that will ensure all method calls jump to valid implementations during program execution, and (3) various optimizations to reduce the SAFEDISPATCH runtime and code



ValidM	m1	m2	m3
A	A::m1 B::m1 C::m1	A::m2 D::m2	A::m3 E::m3
B	B::m1	A::m2	A::m3
C	C::m1	A::m2 D::m2	A::m3 E::m3
D	C::m1	D::m2	A::m3
E	C::m1	A::m2	E::m3

Fig. 4. **Example Class Hierarchy Analysis (CHA).** Our Class Hierarchy Analysis is a static (compile time) analysis that uses the class hierarchy to compute which method implementations can be invoked by objects of each class type. The left diagram above shows an example hierarchy of five classes where subclasses point to their parent class: `D` and `E` are subclasses of `C` while `B` and `C` are subclasses of `A`. These classes have three methods: `m1`, `m2`, `m3`. In each class's box, we denote inheriting a parent's method implementation with `*` and list the names of overridden methods. For example, in this case `C` overrides `A`'s implementation of `m1`, but inherits the implementations of `m2` and `m3`. The results of our Class Hierarchy Analysis (CHA) is the `ValidM` table, specifying for each object type which implementations of a method may be invoked at runtime, according to C++ dynamic dispatch rules. In the example table above right, we see that calling method `m2` on an object statically declared to have type `C` can invoke either class `A`'s or `D`'s implementation of `m2`.

```
// ValidM maps class C and method name N to the set of
// func ptrs implementing N for C and its subclasses
map<class, map<string, set<method>> ValidM;

// computing ValidM at compile time
ValidM = new map<class, map<string, set<method>>();
foreach (class c in all_classes()) {
  ValidM[c] = new map<string, set<method>>();
  // all_method_names(c) returns all method names of class c,
  // including any methods inherited from parent classes
  foreach (string n in all_method_names(c)) {
    ValidM[c][n] = new set<method>();
    // all_subclasses(c) returns c and all its subclasses
    foreach (class sc in all_subclasses(c)) {
      // static_lkup(sc, n) returns the func ptr
      // implementing the method named n for an object of
      // class sc, according to C++ dynamic dispatch rules
      ValidM[c][n].add(static_lookup(sc,n));
    }
  }
}
```

Fig. 5. **Our CHA which constructs `ValidM` at Compile Time.** At compile time SAFEDISPATCH performs CHA to construct `ValidM`, a table specifying for each method of each class type which implementations may legitimately be invoked at runtime. The SAFEDISPATCH compiler generates `ValidM` by iterating over all the program's classes. For each class c, SAFEDISPATCH considers all the names of c's methods, including those transitively inherited from parent classes. For a given method name n, SAFEDISPATCH determines which implementations of n may be invoked at runtime by iterating over all of c's (transitive) subclasses, including c itself. For each subclass sc of c, SAFEDISPATCH determines statically which implementation of n an sc object would invoke and adds it to the set of valid implementations in `ValidM[c][n]`.

size overhead. We describe each of these three passes in more detail below.

A. Class Hierarchy Analysis

SAFEDISPATCH instruments a program to ensure all *runtime* virtual method calls are valid, but before inserting these dynamic checks we must first determine, at *compile time*, which implementations are valid for each virtual method call site. Class Hierarchy Analysis [22] (CHA) is a static analysis that gathers this information by constructing the program's class hierarchy, i.e. immediate subtyping relation, and then traverses this class hierarchy to compute the set of valid implementations for each virtual method of every class. The end result produced by CHA will be a map `ValidM` which gives us, for each class c and each virtual method n, the set `ValidM`$[c][n]$ of method implementations that could be invoked at runtime if an object with static type c were used to call n.

Consider the example CHA results in Figure 4. In this case, the program being analyzed only contains five classes forming a three-layer hierarchy: **D** and **E** are subclasses of **C** while **B** and **C** are subclasses of **A**. Conceptually, this hierarchy is computed by creating a graph containing a node for each class in the program and then adding an edge from class c to c' whenever c extends c'. Each node also stores information about its class's methods, in particular indicating which implementations are inherited from parents (which we depict using `*`) and which the class overrides with its own implementation (which we depict using the method's name).

Our version of CHA analyzes, for each method n of each class c, which of c's subclasses override n with their own implementation. Along with c's (possibly inherited) implementation, the set of such method implementations are the only valid callees that may be invoked by an object of static type c when it calls n at runtime. This is made precise by the code shown in Figure 5, which computes this information and stores the result in a table called **`ValidM`**.

In practice, implementing CHA for large, complex applications like browsers poses a serious challenge, primarily due to subtle interactions between the many C++ inheritance mechanisms, e.g. access modifiers, templates, virtual vs. non-virtual method properties, overloading, and multiple inheritance. To manage this complexity, we build on top of the Clang++ module responsible for constructing C++ vtables at compile time. Clang++ is an industrial strength compiler, capable of handling the tremendous complexity that arises in real-world C++ applications.

Precision and Scalability. SAFEDISPATCH uses CHA to determine, at compile time, which program locations a runtime method call may legitimately jump to. As a type-based analysis, CHA is relatively lightweight and scales up to large, complex applications. However, type-based analyses scale because they are generally coarse-grained and therefore less precise. It is possible that an object x stored in a variable of static type c only ever has runtime type c' where c' is a subclass of c. In such instances, CHA will overestimate the set of valid implementations x may invoke, including the implementation for c and all implementations in subclasses of c, while in reality only the implementation in c' should be called at runtime.

Such sources of imprecision could be remedied by using a more powerful static analysis. The additional precision would

```
// source level method call
o->x(args);

            — — — — — — — — — — — —

// (A) generated code without check inlining
vtable t = *((vtable *)o);
method m = t[vtable_position(x)];
check(static_typeof(o), "x", m);
m(o, args);

// (B) generated code with check partially inlined
vtable t = *((vtable *)o);
method m = t[vtable_position(x)];
if (m != m1 && m != m2 && m != m3)
  check(static_typeof(o), "x", m);
m(o, args);

            — — — — — — — — — — — —

void check(type c, string n, method m) {
  if (!ValidM[c][n].contains(m)) {
    error("bogus method implementation!");
  }
}
```

Fig. 6. **SAFEDISPATCH Instrumentation.** At each method call site, SAFEDISPATCH inserts a check in the generated code to ensure that objects only invoke methods allowed by the static C++ type system. As shown in (A), the basic SAFEDISPATCH instrumentation simply adds a call to the **`check()`** function immediately before the jump to a method implementation. **`check(`**c, n, m**`)`** consults the **`ValidM`** table to ensure that function pointer m is a valid implementation of the method named n for objects with static type c. To avoid an extra function call at every method invocation, SAFEDISPATCH actually uses profiling information to partially inline **`check()`**. As shown in (B), SAFEDISPATCH inserts a branch to test if the function pointer looked up from the calling object's vtable is one of the most common valid implementations of the method used at this call site. If it is, SAFEDISPATCH safely skips the call to **`check()`**, thus avoiding the overhead of an additional function call in the common case. Note that all expressions in *italics* in the code above are evaluated *at compile time* as they require source-level information available only to the compiler.

provide stronger security guarantees by further restricting an attacker's ability to invoke method implementations that should never arise during legitimate program execution. However, accurately tracking which classes flow to a particular variable x at compile time would require a precise whole program dataflow analysis. While such analyses exist, they often don't scale to the kinds of programs we aim to protect, leading to unacceptable increases in compile time. Those analyses that *can* scale in fact do so by giving up on precision, which would bring us back to square one. As a result, we feel that our type-based approach in CHA presents the best tradeoff by being precise enough to prevent real world attacks without dramatically increasing compile times.

We do note that CHA is fundamentally a whole program analysis, and thus requires all an application's code to be available at compile time. Unfortunately, this currently precludes the use of separate compilation in our prototype implementation. However, our SAFEDISPATCH implementation is a research prototype and we feel that future work can address this limitation by annotating compiled object files with partial analysis results and composing those results to complete SAFEDISPATCH's program instrumentation at linktime.

B. SAFEDISPATCH *Method Checking Instrumentation*

After SAFEDISPATCH computes the CHA results, it can instrument the program with checks to ensure that whenever an object calls a virtual method, control jumps to one of the method implementations statically determined to be valid. Figure 6 shows how SAFEDISPATCH instruments each source level method call. For now, consider the basic strategy illustrated in part (A) of Figure 6. In the generated code for `o->x(args)`, after the implementation `m` for method name `"x"` has been looked up in the vtable dereferenced from `o`'s vtable pointer, SAFEDISPATCH inserts a call to **`check(`***static_typeof(o)***`, "x", m)`** *before* invoking **`m`**. This call to **`check`** consults the CHA results in **`ValidM`** to ensure that **`m`** is one of the valid implementations for **`"x"`** when called by an object which has **`o`**'s static type. Note that expressions in *italics* are evaluated *at compile time* as they require source-level information available only to the compiler. As shown in part (B) of Figure 6, SAFEDISPATCH also reduces runtime overhead by partially inlining calls to the **`check`** function, which we discuss in greater detail below.

Data Structures for Checking. The operation for checking method validity, `ValidM[c][n].contains(m)`, is critical for performance since it is inserted at every virtual method call site. Broadly speaking, SAFEDISPATCH uses an array of sets of valid method implementations to perform this validity checking. More specifically, for each pair (c, n) where c is a class and n is a method name, SAFEDISPATCH generates at compile time a unique natural number $i_{(c,n)}$ which is used to index into a large array of sets. The set at position $i_{(c,n)}$, which contains the possible implementations for method n of class c, is represented as an unordered array of pointers to method addresses. Therefore `ValidM[c][n].contains(m)` involves an array lookup to retrieve `ValidM[c][n]`, followed by a linear scan through the resulting set. In our experiments we found that the average set size was very small (1.44 for method checking) and as result we do not expect that using a more elaborate data structure for representing these sets (e.g. a hash-set) would reduce the overhead significantly. Instead, we focus on other aggressive optimizations, for example the inlining of common checks, as explained in Section III-C.

Externalizing Linktime Symbols. One subtlety of the method checking instrumentation is that the compiler does not statically know the concrete address where method implementations will be placed at linktime. It may seem that the SAFEDISPATCH compiler can handle this issue by simply referring to the linktime symbols for each method implementation. However, many modern C++ compilers restrict the linktime symbols for method implementations to only *internal* symbols, meaning that they cannot be referred to outside of code for their class. This poses a problem for SAFEDISPATCH as we need to check method implementation addresses wherever they may be called, not just in the class where they're defined. To address this issue, we *externalize* all linktime symbols for method implementations, allowing us to refer to them outside of their defining class. It would be straightforward to add an additional pass to check that these externalized symbols are only used in (1) internally by the defining class or (2) in SAFEDISPATCH instrumentation, together providing a guarantee equivalent to that of the unmodified C++ compiler.

C. SafeDispatch Optimizations

To minimize SafeDispatch's runtime overhead, we developed a handful of optimizations to reduce the cost of each check. Most importantly, we profile applications and partially inline the checks performed by the **check** function as shown in part (B) of Figure 6. This partial inlining compares the function pointer retrieved from an object's vtable against the concrete addresses of the **N** most common implementations of the method being called in profiling. In Figure 6 we limit **N** to just the three most common implementations, but in practice we can choose a value that balances the performance improvement of inlining against the increase in code size, which, in the worst case, could negatively impact instruction cache performance. In our actual experiments, discussed in Section VI, we inline all checks observed during profiling, which increases codesize, but did not present significant performance overhead for our benchmarks.

SafeDispatch also performs *devirtualization*: in the case that CHA is able to statically determine there is a single valid method implementation at a given method call site, we rewrite the call to forgo vtable lookup and directly call the unique valid implementation. This avoids unnecessary memory operations to load the vtable and other computations to set up a virtual method call.

Now that we have inlined frequently executed checks, the high-level code in part (B) of Figure 6 still needs to be translated into low-level code. A direct naïve translation leaves room for two important optmizations, which we now describe. Consider again the code in part (B) of Figure 6, and let's look at a direct unoptimized translation to low-level code, as shown in part (A) of Figure 7. One source of overhead in this low-level code is that there are *two* opportunities for branch mis-prediction: one is to mis-predict which of the `if (..) goto L1` statements will fire; the second is to mis-predict where the indirect call through `m` will go (note that `m` is a function pointer). Our first low-level optimization is that we can remove the second mis-prediction opportunity by placing a direct call once we know which of the three conditional has fired. This is shown in part (B) of Figure 7, where we now have direct calls for all checks that have been inlined. However, this code now has a lot of code duplication – namely all the setup for parameters. While this doesn't affect the number of instructions executed at run-time, it creates code bloat, which can have adverse effects on instruction-cache performance. Our second low-level optimization is that we hoist the duplicate code from inside the conditionals and use a single copy right before the conditionals, as shown in part (C) of Figure 7.

With all of the above optimizations, namely profile-based inlined checks and low-level optimizations, we were able to reduce the runtime overhead of SafeDispatch to 2.1% and the codesize overhead to 7.5%. Section VI will provide a more detailed empirical evaluation of the overheads of SafeDispatch.

```
// source level method call
o->x(args);

— — — — — — — — — — — —

// (A) direct unoptimized translation
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    if (m == m1) goto L;
    if (m == m2) goto L;
    if (m == m3) goto L;
    check(static_typeof(o), "x", m);
L:  setup_call_args(o, args);
    indirect_call m;
    ...

// (B) eliminate indirect calls
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    if (m == m1) goto L1;
    if (m == m2) goto L2;
    if (m == m3) goto L3;
    check(static_typeof(o), "x", m);
    setup_call_args(o, args);
    indirect_call m;
    goto LR;
L1: setup_call_args(o, args);
    direct_call m1;
    goto LR;
L2: setup_call_args(o, args);
    direct_call m2;
    goto LR;
L3: setup_call_args(o, args);
    direct_call m3;
    goto LR;
LR: ...

// (C) eliminate duplicate code
    vtable t = *((vtable *)o);
    method m = t[vtable_position(x)];
    setup_call_args(o, args);
    if (m == m1) goto L1;
    if (m == m2) goto L2;
    if (m == m3) goto L3;
    check(static_typeof(o), "x", m);
    indirect_call m;
    goto LR;
L1: direct_call m1;
    goto LR;
L2: direct_call m2;
    goto LR;
L3: direct_call m3;
    goto LR;
LR: ...
```

Fig. 7. **Low-level SafeDispatch Optimization.** The code above illustrates low-level optimizations used in SafeDispatch to eliminate branch misprediction for frequently called methods and to eliminate duplicate code for setting up method invocations. As in Figure 6, all expressions in *italics* above are evaluated *at compile time*.

IV. An Alternate Approach: Vtable Checking

The previous section showed how SafeDispatch checks the control flow transfer at virtual method call sites. In this section, we present an alternate technique which establishes the same control-flow guarantee, but provides additional data integrity guarantees in the face of multiple inheritance, at the expense of additional runtime overhead. Later, in Section VI, we evaluate and compare the overhead of both approaches.

A. Pointer Offsets for Multiple Inheritance

To better explain this alternate approach, we first review vtables in more detail. In practice, vtables store more than just function pointers; they also contain offset values that are used to adjust the `this` pointer appropriately in the face of multiple inheritance.

For example, consider a class `C` that inherits from both `A` and `B`. The data layout of `C` objects will first include the fields from `A`, followed by the fields from `B`. Inherited methods from

A will work unmodified on objects of type C because the offset of A's data fields are the same in A as in C. However, methods inherited from B will not work, because B's methods assume that B's fields start at the beginning of the object, whereas in C these fields are located *after* A's fields.

To address this problem, the compiler creates wrappers in C for methods inherited from B. Before calling B's original implementation of the method, the wrapper adjusts the calling object's `this` pointer by an appropriate offset so that it points to the B part of the C object. The situation is further complicated if C is subclassed again using additional multiple inheritance, in which case the layout for the fields inherited from A and B could change in the subclass of C. To address this problem, pointer offsets for `this` are stored in the vtable, so that the correct offset can be used at run-time depending on what class is being used to make the method call.

While our approach from Section III always protects against malicious control flow at virtual method call sites, it does not defend against an attacker counterfeiting a vtable with incorrect `this` pointer offsets. If an attacker successfully mounts such an attack, our previously described approach would still protect the control flow at virtual method calls, but the attacker could corrupt the `this` offset on entry to a method, potentially leading to further data corruption.

B. vtable Checking

To additionally protect `this` pointer offsets at method calls, we implemented an alternate vtable hijacking defense called *vtable checking*. Instead of checking the validity of the function pointer looked up from an object's vtable, we check the vtable pointer itself to ensure that it is valid given the static type of the calling object. In this way, we not only guarantee valid control flow at method calls, but also ensure that the offset value of `this` is computed appropriately.

Figure 8 shows how each source level method call is instrumented in the vtable checking approach. As in Figure 6, expressions in *italics* are evaluated *at compile time* as they require source-level information available only to the compiler. We insert a check similar to the method checking instrumentation shown in Figure 6, but move the instrumentation earlier to check the *vtable* itself instead of the function pointer retrieved from it. In general, for code generated for method call `o->x(args)`, we insert a call to the **`vt_check(`***static_typeof(o)***`, t)`** after vtable `t` has been loaded from `o`'s vtable pointer. This call to **`vt_check`** consults the results of a modified CHA analysis to ensure that **`t`** is one of the valid vtables for an object of **`o`**'s static type. The computation for `ValidVT` is a modified, simpler version of the computation for `ValidM` described in the previous section, since the compiler already computes vtables. In particular, for each class `c` we collect the vtables for `c` and all of its subclasses, and store this entire set in `ValidVT[c]`. Similarly to method checking, the operation `ValidVT[c].contains(t)` is performed in two steps: `ValidVT[c]` is implemented as an array lookup and `contains(t)` is implemented using linear search. Here again, the average size of `ValidVT[c]` in our experiments was very small (2.58) and we reduce runtime overhead by selectively inlining calls to the **`vt_check`** function, taking

```
// source level method call
o->x(args);

  — — — — — — — — — — — —

// generated code with vtable check partially inlined
vtable t = *((vtable *)o);
if (t != t1 && t != t2 && t != t3)
vt_check(static_typeof(o), t);
method m = t[vtable_position(x)];
m(o, args);
  — — — — — — — — — — — —

void vt_check(type c, vtable t) {
  if (!ValidVT[c].contains(t)) {
    error("bogus vtable!");
  }
}
```

Fig. 8. **Alternate SafeDispatch vtable Checking.** The instrumentation above illustrates an alternate vtable hijacking defense: checking the vtable pointer itself *before* using it to look up a method implementation. Similar to the approach shown in Figure 6, the SafeDispatch instrumentation for this alternate strategy inserts a check in the generated code at each method call site, but in this case the check ensures that the calling object's vtable pointer agrees with the static C++ type system. The **`vt_check(`**c, t**`)`** function (analogous to the **`check()`** function discussed earlier) consults the **`ValidVT`** table (constructed from a modified CHA) to ensure that vtable t is a valid vtable for objects of c's static type. As in Figure 6, we partially inline this check using profiling information to avoid the overhead of an extra function call at most method invocations. Again, note that all expressions in *italics* in the code above are evaluated *at compile time* as they require source-level information available only to the compiler. This alternate defense has higher overhead, but provides stronger data integrity guarantees in the face of multiple inheritance.

advantage of profiling information as discussed in the previous section.

C. Performance Implications

The vtable checking approach described above provides a stronger security guarantee than the method checking approach described in the previous section, as it also ensures the integrity of `this` pointer offsets. Unfortunately, this stronger guarantee also incurs higher runtime overhead: since subclasses frequently inherit method implementations from their parent classes, at any virtual method call site, the number of valid vtables is *always* greater than or equal to the number of valid method implementations that can be invoked.

To better understand why this is the case, consider an example in which a class A declares method `foo`, and suppose there are many subclasses of A, none of which override `foo`. Now for any method call `x->foo()` where the static type of `x` is A, method checking just needs to compare against `A::foo`, since it is the only valid implementation of `foo`. On the other hand, vtable checking must compare against each vtable of the many subclasses of A, since each subclass has its own vtable. In practice, we've measured the difference between the number of valid vtables and the number of valid method implementations at a given call site to be roughly a factor of two. We explore the performance implications of this difference further in Section VI.

V. A Hybrid Approach for Method Pointers

In previous sections we described two vtable hijacking defenses, method checking and vtable checking, each presenting

```
class A {
  public: virtual void foo(int) { ... }
};

class B: A {
  public: virtual void foo(int) { ... }
};

void (A::*f)(int); // declare f as ptr to some method of A
f = &A::foo;       // f now points to the foo method

A* a = new A();
(a->*f)(5);        // method call via f ptr, invokes A::foo

a = new B();
(a->*f)(5);        // method call via f ptr, invokes B::foo
```

Fig. 9. **Method Pointer Example.** Because C++ method pointers are invoked via dynamic dispatch, even though `f` is only assigned once, the first call above jumps to `A::foo` while the second jumps to `B::foo`.

different tradeoffs. To best choose between these tradeoffs, we must consider additional subtleties arising from yet another C++ feature: *method pointers*. Conceptually, C++ method pointers are similar to traditional function pointers, except that pointers to virtual methods are *invoked by dynamic dispatch*, which means they could be exploited by vtable hijacking attacks and thus SAFEDISPATCH must also protect virtual calls through method pointers.

Figure 9 illustrates the behavior of C++ method pointers with two simple classes, `A` and `B`, where `A` contains a single method `foo` and `B` extends `A` and overrides `foo`. The method pointer `f` is declared to point to a method of an object of type `A` or one of `A`'s subclasses, and then `f` is assigned to point to `A::foo`. Next an `A` object is allocated and `A::foo` is called through the method pointer `f`. Afterward a `B` object is allocated and the *same method pointer*, `f`, is used to call one of the object's methods. However, in this case, control jumps to `B::foo` instead of `A::foo` since method pointers are *invoked by dynamic dispatch*.

To implement method pointer semantics, C++ compilers generate code which stores a vtable index in method pointers instead of the concrete address of a method's implementation. For example, if `foo` is placed at index `0` in the vtables of `A` and `B`, then the statement `f = &A::foo` will store the value `0` in `f`. When a call is made through a method pointer, the method pointer's value is used to index into the calling object's vtable to retrieve the appropriate method implementation to invoke.

A. Revisiting Previous Approaches

We now evaluate our previous two approaches, method checking and vtable checking, in the face of method pointers. First, consider our vtable checking technique from Section IV. Fortunately, vtable checking correctly handles method pointers with only a slight modification: since a method pointer is simply a vtable index and vtable checking guarantees the validity of vtables at runtime, SAFEDISPATCH simply checks that vtable indices from method pointers are within the valid range of methods for the given class, thus ensuring that method implementations retrieved by indexing into valid vtables with a method pointer will also be valid. While simple, this modification is essential for preventing hijacking attacks through method pointers: if an attacker could arbitrarily set the method index to be out of range for the given class's vtable, they could cause a virtual method pointer call to jump to malicious code.

Second, consider our method checking technique from Section III. In particular, consider a call through a method pointer of the form `(x->*f)(...)`, where the class used in the declaration of method pointer `f` is `C`. We must modify our method checking approach so that for such calls, the instrumentation checks, at runtime, that the function pointer extracted from the calling object's vtable is one of the implementations for *any* method of `C` or its subclasses. This conservative approach can lead to a blow up in the number of required checks for large class hierarchies with many methods, like those found in modern web browsers. This effect is seen in Section VI where we evaluate and further compare our different defenses. Unfortunately, improving on this approach would require a precise whole program dataflow analysis to compute which method implementations a pointer may point to. Despite decades of research, such analyses are very difficult to scale to the large, complex applications most frequently targeted by vtable hijacking attacks.

B. Hybrid Approach

Comparing method checking and vtable checking in the face of method pointers leads to a key observation: at method pointer call sites, vtable checking typically requires many fewer comparisons than method pointer checking, since method pointer checking must compare against all method implementations from several classes. This situation is exactly the opposite from traditional method calls where vtable checking always demands at least as many comparisons as method checking, as discussed at the end of Section IV.

This observation suggests a hybrid approach: perform vtable checking (enhanced with vtable index range checks) at method pointer call sites and method checking at traditional method call sites. We implemented this hybrid approach in SAFEDISPATCH and found that it incurs less runtime overhead than all other techniques, while providing the same strong security guarantees against vtable hijacking. We further discuss the performance implications of our hybrid approach in Section VI. At a member function call site, the numbers of method/vtable checks are compared, and vtable checks are used only when the number of the vtable checks is stricly less than the number of the method checks.

VI. EVALUATION

In this section we evaluate SAFEDISPATCH along three primary dimensions: (A) runtime and code size overhead, (B) effort to develop our prototype, and (C) compatibility with existing applications and programming practice.

A. SAFEDISPATCH *Overhead*

To evaluate the overhead of our SAFEDISPATCH defense, we used our enhanced C++ compiler to build a vtable-safe version of Google Chromium [20], a full-featured, open source web browser which forms the core of the popular Google Chrome browser [19]. Google Chromium is extremely large and complex, far larger than any SPEC benchmark for example. It contains millions of lines of production code, in diverse components (HTML renderer, JPEG decoder, Javascript

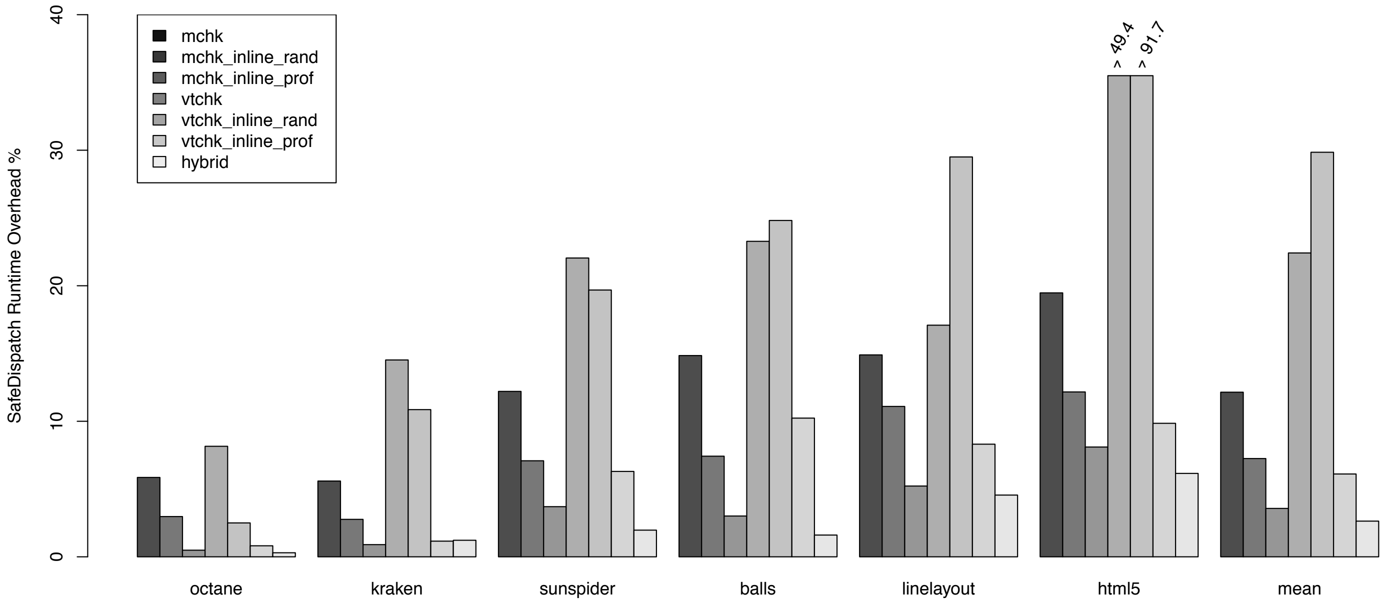


Fig. 10. **SAFEDISPATCH Overhead.** We measured the overhead of SAFEDISPATCH on the Google Chromium browser over six demanding benchmarks: three industry standard JavaScript performance suites (octane, kraken, and sunspider) and three HTML rendering performance tests (balls, linelayout, and html5). All results are reported from the average of five runs, using percentage overhead compared to a baseline with no instrumentation. "mchk" is the unoptimized method pointer checking from Section III, "vtchk" is the unoptimized vtable checking from Section IV. "inline_rand" indicates that we inline all checks that our Class Hierarchy Analysis tells us are needed for safety, but we inline them in a random order (i.e. no profile information). "inline_prof" indicates that we inline the checks observed during profiling in order of how frequently they occur. "hybrid" is the hybrid approach from Section V, which does profile-based inlining, but also combines method pointer checking and vtable checking. Note that two bars did not fit in the graph with the scale we chose for the y axis, namely "vtchk" and "vtchk_inline_rand" for html5; we shortened those bars, and show their values right on top of the bars (rather than change the scale and make all the other bars more difficult to read).

JIT, IPC library, etc.) developed across multiple organizations (Google and various open source groups). Chromium serves as an ideal test case for SAFEDISPATCH: not only is it a complex, high performance C++ application with millions of users, but has also been targeted by several vtable hijacking attacks [12], [9].

Benchmarks. We measured SAFEDISPATCH overhead on Chromium over six demanding benchmarks: three industry-standard JavaScript performance suites (*octane* [23], *sunspider* [24], and *kraken* [25]), and three HTML rendering performance tests (*balls*, *linelayout*, and *html5*). The three HTML rendering benchmarks are drawn from the WebKit performance test suite [26], the engine underlying several major web browsers including Google Chrome, Apple Safari, and Opera. We selected these benchmarks from the suite as three of the most important for performance and rendering correctness. We briefly describe the benchmarks below:

octane, *kraken*, and *sunspider* are the JavaScript performance benchmarking suites from the Google Chrome, Mozilla Firefox, and Apple WebKit teams respectively. These benchmarks strive to measure real-world workloads and exercise the most important browser functionality, while remaining statistically sound and pushing for improvement on bleeding-edge features. For octane we report the benchmark score where higher is better and for kraken and sunspider we measure running time in milliseconds where smaller is better.

balls creates thousands of small ball-shaped DOM elements, moves them around on the screen, measures how many of them can be moved in a fixed amount of time, and reports frames per second as its output. We report frames per second (fps); higher is better.

linelayout creates multiple DOM objects containing copious text. The renderer must draw many text lines, automatically inserting line breaks and allocating DOM objects efficiently on the screen, ensuring the renderer correctly handles the layout of DOM elements on the screen. We report number of complete runs in a fixed period; higher is better.

html5 performs millions of DOM manipulations to test numerous HTML5 features and is one of the most demanding WebKit performance tests. Each complex rendering is compared to an industry-standard reference rendering, thus ensuring optimizations have not introduced incorrect behavior. We report timing results in milliseconds; smaller is better.

Runtime Overhead. Figure 10 presents the runtime overhead percentage of SAFEDISPATCH on benchmarks using a number of different approaches and optimizations, whereas Figure 11 presents the raw numbers, including memory overhead. See the caption of Figure 10 for what each configuration of SAFEDISPATCH corresponds to (e.g., "mchk_inline_rand"). All our results are the average of five runs on an otherwise quiescent system running Ubuntu 12.04 on an Intel i7 Quad Core machine with 8GB of RAM.

From Figure 10, we can see that in general, all the "mchk" overheads are smaller than the "vtchk" overheads. This is consistent with the fact that, as described in Section 8, the number of valid vtables at a given method callsite is often 2x greater than the number of valid method implementations. Figure 10 shows the effectiveness of partial inlining of checks

Instrumentation	octane (score)	kraken (ms)	sunspider (ms)	balls (fps)	linelayout (runs)	html5 (ms)	Code Size Overhead %
none	15353	1556	254	14.95	106.26	3543	-
mchk	14454	1643	285	12.73	90.43	4233	7.20
mchk_inline_rand	14897	1599	272	13.84	94.47	3974	14.11
mchk_inline_prof	15278	1570	263	14.50	100.71	3830	7.48
vtchk	14101	1782	310	11.47	88.10	5294	7.31
vtchk_inline_rand	14969	1725	304	11.24	74.91	6793	44.18
vtchk_inline_prof	15228	1574	270	13.42	97.43	3892	7.85
hybrid	15299	1570	256	14.71	102.39	3721	7.48

Fig. 11. **SAFEDISPATCH Benchmarking Results and Code Size Overhead.** The table above shows our benchmarking measurements for SAFEDISPATCH which correspond to the runtime overhead graph in Figure 10: for times reported in milliseconds, smaller is better, for other reported quantities (score, fps, and runs), larger is better. We additionally measured average code size increase due to SAFEDISPATCH data structures and instrumentation, and observed overheads typically well under 10%.

	Benchmark Overhead %		
Profile	octane	kraken	sunspider
octane	0.30	2.51	6.30
kraken	0.79	1.22	6.69
sunspider	1.15	2.25	1.97

Fig. 12. **Cross Profiling.** To evaluate the effect of profiling across benchmarks, we measured the overhead of running each binary optimized for one JavaScript performance suite on the other suites. The numbers reported are percentage overhead for the hybrid approach.

Component	Framework	Language	LOC
Basic Instrumentation	Clang++	C++	177
Class Hierarchy Analysis	-	Python	691
Inlining Optimizations	LLVM	C++	381
Total			1249

Fig. 13. **SAFEDISPATCH Prototype LOC.** The table above characterizes the major components in our SAFEDISPATCH implementation. The basic instrumentation module is implemented as a Clang++ compiler pass and inserts calls to the `check()` fucntion as described in Section III function at each method call site, additionally logging some type data. These logs are used by the CHA module, written in Python, to build the `ValidM` and `ValidVT` used during checking at runtime. The final module is implemented as a set of low-level LLVM passes to inline checks based on profiling information.

not just using profile information, but also using a random order. The random order is meant to capture the situation where we perform inlining, but we don't have profile information. We can see that inlining alone, without profile information ("mchk_inline_rand" and "vtchk_inline_rand") improves performance compared to the unoptimized instrumentation, but only for method checking. For vtable checking, the random-order inlining causes a slowdown because there were too many checks to inline, which affected performance negatively (this is confirmed by the memory overhead shown in Figure 11. Inlining with profile information ("mchk_inline_prof" and "vtchk_inline_prof") provides a significant reduction in percentage overhead compared to the unoptimized instrumentation. Finally, Figure 10 also shows that that the hybrid approach from Section V has the lowest overhead by far, about 2% on average.

Cross Profiling. As shown above, profiling information can significantly reduce SAFEDISPATCH overhead. However, once deployed, applications are often run on inputs that were not profiled. To measure the effectiveness of profiling on one application and running on another, we used each of the binaries optimized for each JavaScript benchmark and ran it on the others. We focused on JavaScript benchmarks for this cross-profiling evaluation because the rendering benchmarks each evaluate a different kind of rendering (e.g. text, graphics, html rendering), and it would be unlikely that one of them would be a good predictor for others (in essence we would have to profile all three rendering benchmarks to get a representative set, but then this would not evaluate cross-profiling). Figure 12 shows the results of cross-profiling for the hybrid approach. Each row and each column is a benchmark, and at row y and column x, we show the percentage overhead of running the x benchmark using the binary optimized for y's profile information. While we can see that in some cases the overhead jumps to 6%, if we profile with *sunspider*, the overhead still remains in the vicinity of 2%. This may indicate that *sunspider* is a more representative Javascript benchmark, which is better suited for generating good profile information.

Code Size Overhead. We also measured the increase to code size resulting from SAFEDISPATCH data structures and instrumentation in the generated executable, shown in the final column of the table from Figure 11. For the hybrid approach, the generated executable size was within 10% of the corresponding unprotected executable. Note that the memory overhead for "vtchk_inline_rand" is substantial, which is consistent with the run-time overhead for "vtchk_inline_rand" from Figure 10.

B. Development Effort

Our prototype implementation of SAFEDISPATCH has three major components: (1) the basic instrumentation compiler pass,

(2) CHA analysis to generate the `ValidM` and `ValidVT` internal SAFEDISPATCH checking data structures, and (3) inlining optimizations. The size of each component is listed in Figure 13.

The basic instrumentation pass is implemented as a pass in Clang++ while the compiler has access to source-level type information which is erased once a program is translated into the lower level LLVM representation. This pass also produces information used in our second major component, the CHA analysis, which we implemented in a set of Python scripts to build the intermediate `ValdiM` and `ValidVT` tables. Finally, we implemented our inlining passes as an optimization in LLVM which can take advantage of profiling information to order checking branches by how frequently they were taken in profile runs.

C. Compatibility

In principle, SAFEDISPATCH only incurs minimal compile time overhead to build the `ValidM` and `ValidVT` tables and instrument virtual method call sites as described in Sections III, IV and V. Thus, the programmer should be able to use SAFEDISPATCH on every compilation without disrupting the typical edit, compile, test workflow. However, in our current prototype implementation, SAFEDISPATCH performs two full compilations to gather necessary analysis results before instrumenting the code, leading to a roughly 2x increase in compile time. As mentioned above, this is an artifact of our prototype implementation which can easily be fixed and is not an inherit limitation of SAFEDISPATCH.

The SAFEDISPATCH prototype also requires a whole-program CHA to perform instrumentation, and does not currently support separate compilation. There are two main challenges in supporting separate compilation. The first challenge is to make CHA modular. In particular, the compiler would have to generate CHA information per-compilation unit, which the linker would then combine into whole-program information. This approach to CHA is very similar to the approach taken in GCC's vtable verification branch [27], [28], more details of which are discussed in Section VIII. The second challenge is to inline checks in a modular way. In particular, editing code in one file could require additional checks in *another* file. To address this challenge, the compiler could insert calls to **`check`** at compile time, and then replace these calls with inserted inlined checks at link-time (similarly to link-time inlining of function calls). Finally, profiling data for inlining optimizations can be collected using a profile build in which the **`check`** function collects the required function/vtable pointers. This profile build can easily support separate compilation, as it does not require inlining or CHA.

VII. SAFEDISPATCH SECURITY ANALYSIS

In this section we consider the security implications of SAFEDISPATCH including the class of attacks SAFEDISPATCH prevents and some limitations of our approach.

A. SAFEDISPATCH *Guarantee*

The instrumentation inserted by the SAFEDISPATCH compiler guarantees that each virtual method call made at runtime jumps to a valid implementation of that method according to C++ dynamic dispatch rules. This guarantee immediately eliminates an attacker's ability to arbitrarily compromise the control flow of an application using a vtable hijacking attack. Our defense would prevent crucial steps in many recent, high profile vtable hijacking attacks, e.g. Pinkie Pie's 2012 Zero-day exploit of Google Chrome which escaped the tab sandbox and allowed an adversary to compromise the underlying system. In addition to preventing many attacks, SAFEDISPATCH provides an intuitive guarantee in terms of the C++ type system, which is easy to understand for programmers who are familiar with the type system. Furthermore, the programmer cannot inadvertently nullify the SAFEDISPATCH guarantee through a programming mistake; the checks inserted by SAFEDISPATCH will detect errors such as incorrect type casts which would otherwise lead to a method call invoking an invalid method implementation.

The SAFEDISPATCH guarantee provides strong defense against vtable hijacking attacks, regardless of how the attack is mounted, e.g. use-after-free error, heap based buffer overflow, type confusion, etc. As discussed further in the next section on related work, other defenses only focus on particular styles of attack (for example mitigating use-after-free errors by reference counting), or incur non-trivial overhead (for example using a custom allocator to ensure the memory safety properties necessary to prevent vtable hijacking). Furthermore, SAFEDISPATCH protection is always safe to apply: all programs should already satisfy the SAFEDISPATCH guarantee – we are simply enforcing it.

SAFEDISPATCH also defends against potentially exploitable, invalid typecasts made by the programmer [29]. If a programmer incorrectly casts an object of static type c to another type c' and at runtime the object does not have type c', then methods invoked on the object will not be valid implementation and SAFEDISPATCH will signal an error.

The astute reader may wonder why the checks inserted by SAFEDISPATCH instrumentation are any more secure than the vtable pointer stored in a runtime object. Unlike such heap pointers, the checks inserted by SAFEDISPATCH and their associated data structures are embedded in the generated executable which resides in *read-only memory*, ensuring that an attacker will not be able to corrupt SAFEDISPATCH inserted checks at runtime. Of course, this assumes the attacker will not be able to remap the program's text segment, or portion of memory containing the application's executable code, to be writable.

B. SAFEDISPATCH *Limitations*

SAFEDISPATCH guarantees that *one of* the valid method implementations for a given call site will be invoked at runtime, *not* that the correct method will be called. For example, an attacker could still corrupt an object's vtable pointer to point to the vtable of a child class, causing an object to invoke a child class's implementation of a method instead of it's own. While this call would technically satisfy the static C++ dynamic dispatch rules, it could lead to further memory corruption or other undesirable effects. However, we are not aware of any exploits in the wild which take advantage of such behavior.

SAFEDISPATCH detects vtable pointer corruption precisely when it would result in an invalid method invocation. This does not prevent other memory corruption attacks, such as overwriting the return address stored in a function's activation record on the stack. SAFEDISPATCH also does not currently prevent corrupting arbitrary (non-object) function pointer values. Such function pointers are important in systems making extensive use of callbacks or continuations. SAFEDISPATCH could be extended to protect such calls through function pointers by conceptually treating them as method invocations of a special ghost class introduced by the compiler. This change, which we will explore in future work, would also be transparent to the programmer and would further strengthen our guarantee.

SAFEDISPATCH only protects the code it compiles. Thus, if an application dynamically loads *unprotected* system libraries, an attacker may be able to compromise control flow within the library code via vtable hijacking. While such libraries can be compiled with SAFEDISPATCH to prevent such attacks, it's important to note that SAFEDISPATCH requires performing a whole program Class Hierarchy Analysis on the *entire program*, including all application libraries *and* all system libraries. Unfortunately, it is well known that such whole program analyses present challenges in the face of separate compilation, dynamically linked libraries, and shared libraries. As a result, our current SAFEDISPATCH prototype protects the entire application code, including all application libraries, but it does not protect shared system libraries such as the C++ standard library.

Dynamically linked libraries are also a possible source of incompatibility with the current SAFEDISPATCH prototype. For example, consider an application that uses a subclass implemented in an external, dynamically linked library. Since the subclass information is not statically available to SAFEDISPATCH's CHA, any such dynamically loaded subclass method implementations will be reported as invalid by `check` at runtime. To overcome this limitation, SAFEDISPATCH would be required to dynamically update its `ValidM` and `ValidVT` tables as dynamic libraries are loaded at runtime by instrumentation of certain system calls (e.g., `dlopen`). In future work, we hope to address this limitation by developing better techniques for performing our CHA analysis in the face of separate compilation and dynamically linked libraries.

C. *Performance and Security Tradeoffs*

As discussed in previous sections, there are multiple strategies for enforcing the SAFEDISPATCH guarantee which lead to different security and performance tradeoffs. Vtable checking provides additional data integrity guarantees over method checking, in particular for `this` pointer offsets in the face of multiple inheritance, but at the cost of additional runtime overhead. Our hybrid approach adopts vtable checking at method pointer call sites to reduce runtime overhead, but uses method checking at non-method-pointer call sites, and so does not provide the same data integrity guarantees as vtable checking. Although the additional data integrity guarantee provided by vtable checking may mitigate some attacks, we feel that the significantly reduced overhead of our method checking and hybrid approaches offer a more realistic tradeoff for complex, high performance applications like web browsers.

VIII. RELATED WORK

The research community has developed numerous defenses to increase the cost of mounting low-level attacks that corrupt control data, steadily driving attackers to discover new classes of exploitable programming errors like vtable hijacking. In this section we survey the existing defenses most relevant to vtable hijacking, consider their effectiveness at mitigating such attacks, and compare them to SAFEDISPATCH.

Reference Counting. Reference counting [30], [31], [32] is a memory management technique used in garbage collectors and complex applications to track how many references point to an object during program execution. When the number of references reaches zero, the object may safely be freed. Use-after-free errors can be avoided using reference counting by checking that an object has a non-zero number of references before calling any methods with the object. While this may help increase the attack complexity of vtable hijacking attacks mounted by exploiting use-after-free bugs, reference counting can have a non-trivial run-time overhead, and it also makes reclaiming cyclic data-structures complicated. Most importantly, however, reference counting cannot fundamentally prevent such attacks. In reference counting, the number of references to an object is stored in the heap, and thus an adversary capable of corrupting vtable pointers would also be able to corrupt reference counts, thereby circumventing any reference counting based defense. In contrast, SAFEDISPATCH instrumentation is placed in the program binary which resides in read-only memory and thus is not susceptible to corruption by an attacker.

Memory Safety. Programs written in *memory safe* languages are guaranteed, by construction, to be free of exploitable, low-level memory errors. This kind of memory safety guarantee is clearly stronger than the guarantee that SAFEDISPATCH provides. However, unfortunately programs written in such languages often suffer significant performance overhead from runtime checking to ensure that all memory operations are safe. This overhead is sufficient to preclude the use of memory safe languages in many performance critical applications. In contrast, SAFEDISPATCH provides strong security guarantees without any assumptions about memory safety and incurs only minimal overhead.

There has also been extensive research on C compilers which insert additional checks or modify language features to ensure memory safety, for example CCured [33], [34], Cyclone [35], Purify [36], and Deputy [37]. While these techniques can help prevent vtable hijacking, they often require some amount of user annotations, and even if they don't, their run-time overheads are bigger than SAFEDISPATCH, especially on large-scale applications like Chrome.

Control Flow Integrity. Control flow integrity (CFI) is a technique that inserts sufficient checks in a program to ensure that every control flow transfer jumps to a valid program location [13]. Recent advances have greatly reduced the overhead of CFI, in some cases to as low as 5%, by adapting efficient checks for indirect targets [14], using static analysis [15], harnessing further compiler optimizations from a trustworthy high-level inline-reference monitor representation [16], or incorporating low-level optimizations [17]. The main difference between our work and these previous CFI approaches lies in the particular point in design space that we chose to explore. Broadly speaking, previous CFI approaches are designed to secure all indirect jumps whereas we focus specifically on protecting C++ dynamic dispatch, which has become a popular target for exploits. In this more specific setting, we provide stronger guarantees than recent CFI approaches while incurring very low performance overhead.

VTable Hijacking Prevention. The GCC compiler has recently been extended with a promising new "vtable verification" feature developed by Google [27], [28], concurrently and indenpendently from SafeDispatch. The GCC approach compiles each C++ source file to an object file extended with local vtable checking data, and the local checking data is combined at load-time into a program-wide checking table. Each virtual method call site is then instrumented with a call to a checking function which uses the program-wide table to determine if the control-flow transfer should be allowed. In many respects, the GCC approach is roughly equivalent to our unoptimized vtable checking approach. In this light, our work extends GCC's approach in the following ways: (1) we explore and empirically evaluate not only vtable checking, but also method checking (2) through this evaluation, we discover and propose a new optimization opportunity in the form of a hybrid approach and (3) we inline common checks. In our implementation, vtable checking without inlining (which is roughly what GCC does) leads to an overhead of about 25%. Through optimizations 2 and 3 above, we reduce the overhead to only 2%. On the other hand, the GCC approach supports separate compilation much more easily than our approach, which requires whole program analysis and profiling.

Another technique for preventing vtable hijacking is VTGuard [38], a feature of the Visual Studio C++ compiler. This approach inserts a secret cookie into each vtable and checks the cookie before the vtable is used at runtime. While this approach has very low performance overhead, it is less secure than ours: the attacker can still overwrite a vtable pointer to make it point to *any* vtable generated by the compiler, something we prevent. Moreover, if the secret cookie is revealed through an information disclosure attack, then the VTGuard protection mechanism can be circumvented.

Memory Allocators and Dynamic Heap Monitoring. Dynamic heap monitoring, like that used in Undangle [39] and Valgrind [40], can help discover memory errors during testing, but are not suitable for deployment as they can impose up to 25x performance overhead, which is unacceptable for the applications we aim to protect. The DieHard [3], [41] custom memory manager has proven effective at providing probabilistic guarantees against several classes of memory errors, including heap-based buffer overflows and use-after-free errors by randomizing and spreading out the heap. While DieHard overhead is often as low at 8%, it demands a heap *at least* 2x larger than what the protected application would normally require, which is unacceptable for the applications we aim to protect. Furthermore, large applications like a browser often use multiple custom memory allocators for performance, whereas DieHard requires the entire application to use a single allocator.

Data Execution Prevention (DEP). After an adversary has compromised program control flow, they must arrange for their attack code to be executed. DEP [2] seeks to prevent an attacker from writing malicious shellcode directly to memory and then jumping to that code. Conceptually every memory page is *either* writable *or* executable, but *never* both. DEP can mitigate vtable hijacking after the attack has been mounted by preventing the attacker from executing code they've allocated somewhere in memory. However, attackers can still employ techniques like Return Oriented Programming [42] (ROP) to circumvent DEP after control flow has been compromised from a vtable hijacking attack. DEP is also often disabled for JIT. While DEP tries to mitigate the damage an attacker can do after compromising control flow, SafeDispatch seeks to prevent a class of control flow compromises (those due to vtable hijacking) from arising in the first place.

Address Space Layout Randomization (ASLR). Like DEP, ASLR [43] seeks to severely limit an attackers ability to execute their attack code *after* control flow has been compromised. It does this by randomly laying out pages in memory so that program and library code will not reside at predictable addresses, making it difficult to mount ROP and other attacks. Unfortunately, for compatibility, many prevalent, complex applications are still forced to load key libraries at predictable addresses, limiting the effectiveness for ASLR in these applications. SafeDispatch helps secure such applications by preventing vtable-hijacking-based control flow compromises from arising in the first place.

IX. Conclusion

Robust vtable hijacking attacks are increasingly common, as seen in sophisticated, high profile attacks like Pinkie Pie's recent exploits of the Chrome browser [12]. In this paper, we addressed the growing threat of vtable hijacking with SafeDispatch, an enhanced C++ compiler to ensure that control flow transfers at method invocations are valid according to the static C++ semantics.

SafeDispatch first performs class hierarchy analysis (CHA) to determine, for each class c in the program, the set of valid method implementations that may be invoked by an object of static type c, according to C++ semantics. SafeDispatch then uses the information produced by CHA to instrument the program with dynamic checks, ensuring that, at runtime, all method calls invoke a valid method implementation according to C++ dynamic dispatch rules.

To minimize performance overhead, SafeDispatch performs optimizations to inline and order checks based on profiling data and adopts a hybrid approach which combines method checking and vtable checking. We were able to reduce runtime overhead to just 2.1% and memory overhead to just 7.5% in the first vtable-safe version of the Google Chromium browser which we built with the SafeDispatch compiler.

We believe that these results are a solid first step towards hardening method dispatch against attack, and that they provide a good foundation for future exploration in this space, including ways of handling separate compilation, and additionally protecting indirect control flow through arbitrary functions pointers.

Acknowledgment

We would like to thank Hovav Shacham and Stephen Checkoway for clarifying the importance of vtable hijacking attacks in the initial stage of the project. We would also like to thank the anonymous reviewers for helping us improve our paper. This work was supported in part by the National Science Foundation through grants 1228967 and 1219172.

REFERENCES

[1] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, 1998.

[2] Microsoft, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," http://support.microsoft.com/kb/875352.

[3] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[4] rix, "Smashing c++ vptrs," http://www.phrack.org/issues.html?issue=56&id=8, 2000.

[5] O. Vertanen, "Java type confusion and fault attacks," in *FDTC*, 2006.

[6] D. Dewey and J. Giffin, "Static detection of c++ vtable escape vulnerabilities in binary code," in *NDSS*, 2012.

[7] Microsoft, "Vulnerability in Internet Explorer could allow remote code execution," http://technet.microsoft.com/en-us/security/advisory/961051, 2008.

[8] H. D. Moore, "Microsoft Internet Explorer data binding memory corruption," http://packetstormsecurity.com/files/86162/Microsoft-Internet-Explorer-Data-Binding-Memory-Corruption.html, 2010.

[9] Google, "Heap-use-after-free in WebCore (exploitable)," https://code.google.com/p/chromium/issues/detail?id=162835, 2012.

[10] Symantec, "Microsoft Internet Explorer virtual function table remote code execution vulnerability," http://www.symantec.com/security_response/vulnerability.jsp?bid=54951, 2012.

[11] VUPEN, "Exploitation of Mozilla Firefox use-after-free vulnerability," http://www.vupen.com/blog/20120625.Advanced_Exploitation_of-Mozilla_Firefox_UaF_CVE-2012-0469.php, 2012.

[12] C. Evans, "Exploiting 64-bit Linux like a boss," http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html, 2013.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[14] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity & randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy, San Francisco, CA*, 2013.

[15] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 29–40.

[16] B. Zeng, G. Tan, and Ú. Erlingsson, "Strato-a retargetable framework for low-level inlined-reference monitors," in *USENIX Security Symposium*, 2013.

[17] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *USENIX Security Symposium*, 2013.

[18] C. Grier, S. Tang, and S. T. King, "Secure web browsing with the op web browser," in *IEEE Security and Privacy*, 2008.

[19] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," in *CACM*, 2009.

[20] A. Barth, C. Jackson, and C. Reis, "The security architecture of the chromium browser," in *Technical Report*, 2008.

[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[22] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP*, 1995.

[23] Google, "Octane javascript benchmark suite," https://developers.google.com/octane/, 2013.

[24] Apple, "Sunspider 1.0 javascript benchmark suite," https://www.webkit.org/perf/sunspider/sunspider.html, 2013.

[25] Mozilla, "Kraken 1.1 javascript benchmark suite," http://krakenbenchmark.mozilla.org/, 2013.

[26] WebKit, "Rendering performance tests," https://code.google.com/p/webkit-mirror/source/browse/PerformanceTests/, 2013.

[27] C. Tice, "Improving function pointer security for virtual method dispatches," http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf, 2012.

[28] ——, "Gcc vtable security hardening proposal," http://gcc.gnu.org/ml/gcc-patches/2012-11/txt00001.txt, 2012.

[29] D. Dewey and J. T. Giffin, "Static detection of c++ vtable escape vulnerabilities in binary code," in *NDSS*, 2012.

[30] G. E. Collins, "A method for overlapping and erasure of lists," in *CACM*, 1960.

[31] D. J. Roth and D. S. Wise, "One-bit counts between unique and sticky," in *ISMM*, 1998.

[32] Y. Levanoni and E. Petrank, "An on-the-fly reference-counting garbage collector for java," in *TOPLAS*, 2006.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy software," in *TOPLAS*, 2005.

[34] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," in *PLDI*, 2003.

[35] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *USENIX ATEC*, 2002.

[36] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of c programs," in *SIGSOFT*, 2004.

[37] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *ESOP*, 2007.

[38] M. R. Miller and K. D.Johnson, "Using virtual table protections to prevent the exploitation of object corruption vulnerabilities," http://patentimages.storage.googleapis.com/pdfs/US20120144480.pdf, 2010.

[39] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA*, 2012.

[40] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[41] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *CCS*, 2010.

[42] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[43] P. Team, "Pax address space layout randomization," http://pax.grsecurity.net/docs/aslr.txt.