DataShield: Configurable Data Confidentiality and Integrity

Scott A. Carr
carr27@purdue.edu
Purdue University

Mathias Payer
mathias.payer@nebewelt.net
Purdue University

ABSTRACT

Applications written in C/C++ are prone to memory corruption, which allows attackers to extract secrets or gain control of the system. With the rise of strong control-flow hijacking defenses, non-control data attacks have become the dominant threat. As vulnerabilities like HeartBleed have shown, such attacks are equally devastating.

Data Confidentiality and Integrity (DCI) is a low-overhead non-control-data protection mechanism for systems software. DCI augments the C/C++ programming languages with annotations, allowing the programmer to protect selected data types. The DCI compiler and runtime system prevent illegal reads (confidentiality) and writes (integrity) to instances of these types. The programmer selects types that contain security critical information such as passwords, cryptographic keys, or identification tokens. Protecting only this critical data greatly reduces performance overhead relative to complete memory safety.

Our prototype implementation of DCI, DataShield, shows the applicability and efficiency of our approach. For SPEC CPU2006, the performance overhead is at most 16.34%. For our case studies, we instrumented mbedTLS, astar, and libquantum to show that our annotation approach is practical. The overhead of our SSL/TLS server is 35.7% with critical data structures protected at all times. Our security evaluation shows DataShield mitigates a recently discovered vulnerability in mbedTLS.

1. INTRODUCTION

Code written in low level languages such as C/C++ comprises the majority of software that modern systems run. Programmers choose these languages for the speed and control they provide. These benefits come with a cost: protecting programs against memory and type safety errors is left to the programmer. In particular, the lack of memory and type safety, and the resulting memory corruptions, has lead to an unending stream of security vulnerabilities.

Preventing memory vulnerabilities in C/C++ code is well explored, but widely adopted protection mechanisms focus on control-flow hijack attacks, neglecting non-control-data attacks. In a control-flow hijack attack, the attacker diverts the program's intended control-flow. However, in non-control-data attacks, the program execution follows a valid path through the program, but the data is attacker-controlled.

Mature control-flow defenses – such as Stack Cookies [12], Address Space Layout Randomization (ASLR) [44], DEP [52], and Control-flow Integrity (CFI) [1] – are widely deployed. Next generation control-flow protection mechanisms such as CFI and Code-Pointer Integrity [27] are widely researched and are being deployed in production systems. For example, Microsoft's Control-Flow Guard [29] and LLVM-CFI [11] are available in production compilers. As these defenses improve, attackers will follow the path of least resistance and shift to non-control-data attacks, which are not prevented by any of the above mechanisms. As the high-profile HeartBleed bug showed [16], non-control-data attacks are as harmful as control-flow hijack attacks [9, 20, 24]. In fact, the Control-Flow Bending [8] non-control-data attack can achieve Turing-complete computation even under CFI.

Complete memory safety mechanisms stop both control-flow hijack attacks and non-control-data attacks, but have not been widely adopted. CCured [35], Cyclone [26], WIT [3] and SoftBound [34] are all different approaches that retrofit C/C++ with some form of memory safety, but (i) impose prohibitively high performance overhead and (ii) may run into compatibility issues with legacy code. In general, determining if a C/C++ program is memory safe is undecidable, so any complete protection mechanism must fall back, at least in part, to runtime checks. This requirement puts an intrinsic lower bound on the overhead of any complete memory protection mechanism. For this reason, we argue that program-wide precise memory protection imposes too much overhead for wide adoption.

We introduce Data Confidentiality and Integrity (DCI) to address these challenges. DCI limits the performance overhead by tagging data as either sensitive or non-sensitive, enforcing precise spatial and temporal safety checks only on sensitive data. For non-sensitive data, DCI only requires coarse bounds and imprecise temporal safety. Coarse checks are low overhead, avoiding expensive metadata lookups. The programmer specifies which data is fully protected and the DCI compiler and runtime enforces the security policy. In this way, DCI allows the programmer to control the overhead/protection trade-off. The DCI policy is:

(i) *Memory safety for sensitive data:* Pointers to sensitive objects can only be dereferenced if they point within the bounds of the associated (valid) memory object.

(ii) *Sandboxing for non-sensitive data:* Pointers to non-sensitive objects can be dereferenced if they point anywhere except a sensitive memory object.

(iii) *No data-flow between sensitive and non-sensitive data:* Explicit data-flow between sensitive memory objects and non-sensitive memory objects is forbidden. Sensitive and non-sensitive objects must reside in disjoint memory regions.

This policy provides spatial and temporal memory safety for sensitive data at runtime, and ensures both confidentiality and integrity of the sensitive data.

Our implementation of DCI, DataShield, consists of three key parts. First, DataShield provides a language the programmer uses to specify protection. The language is a small

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ISBN 978-1-4503-4944-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3052973.3052983

set of annotations that are added to existing C/C++ code. Second, compiler instrumentation identifies the sensitive variables and associated data-flows, and rewrites the program with additional security checks. Third, the runtime system creates and maintains the metadata. Our contributions are:

- Definition of a new security policy we call Data Confidentiality and Integrity;
- An open-source compiler-based prototype implementation of the policy called DataShield with instrumented C/C++ standard libraries;
- Two case studies on the practicality DataShield;
- An instrumented version of an SSL/TLS library (mbedTLS) with sensitive data protection built-in;
- A security evaluation that shows DCI mitigates CVE-2015-5291, a recent vulnerability in mbedTLS.

2. BACKGROUND AND MOTIVATION

In the following sections we: (i) explain the relationship between memory safety, integrity, and confidentiality, (ii) discuss the performance overhead of memory safety, and (iii) show that non-control-data attacks are not prevented by existing low-overhead defense mechanisms.

2.1 Safety, Integrity, and Confidentiality

A program is *memory safe* if it is free from memory access errors, including but not limited to: (i) buffer overflows, (ii) dangling pointers, (iii) null pointer dereferences, (iv) use of uninitialized memory, and (v) double frees [23, 50].

A spatial memory error occurs when an out of bounds pointer (usually the result of incorrect pointer arithmetic) is dereferenced. A temporal memory error occurs when a pointer to uninitialized memory or to freed memory is dereferenced. We will refer to spatial memory safety and temporal memory safety as simply "memory safety".

Data integrity and confidentiality are ensured by enforcing memory safety for both reads and writes through pointers. An integrity violation occurs when an attacker uses a pointer to write outside the intended memory object, while a confidentiality violation occurs when the attacker uses a pointer to read outside the intended memory object.

2.2 Memory Safety Overhead

The absence of memory errors is, in general, undecidable statically for C/C++. Any mechanism guaranteeing memory safety must (at least partially) fall back on dynamic checks and maintain metadata. While several mechanisms have been proposed with varying efficiency, they all require a runtime component that imposes some overhead [30]. Table 1 shows the average overhead of four existing complete safety mechanisms (as reported by the authors). The purpose of the table is not to compare the techniques with each other, but to show there is significant overhead for each mechanism. Straightforward comparison is impossible, since they use different benchmarks, computer architectures, software versions, and operating systems.

Protection	Benchmarks	Avg. Overhead (%)
CCured	SPECINT95	81.75
CCured	Olden	44.33
Cyclone	N/A	38.25
SoftBound	SPEC2000, Olden	67

Table 1: Performance Overhead of Existing Memory Safety Mechanisms (as reported by the authors).

2.3 Non-control-data Attacks

To date, security researchers and attackers have primarily focused on control-flow hijack attacks. Such attacks were effective and simple to execute before protection mechanisms were in place. Stack Cookies [13], ASLR [44], and DEP [52] enforce code integrity and give probabilistic guarantees against code reuse attacks. The adoption of these defenses mitigated simple attacks, e.g., stack smashing or shell code injection. Attackers responded by moving to code reuse attacks, such as return-oriented programming [6, 36]. Following the same pattern, when CFI (or CPI) mechanisms [25, 27, 37, 39, 40, 45, 51, 53] become widely deployed, attackers will shift from control-flow hijack attacks to the next available form of attack, i.e., non-control-data attacks.

Non-control-data attacks [8, 20, 24] are harder to detect because only the data differs between benign and malicious executions. For example, to exploit the HeartBleed bug, the attacker sends a malicious request. For a benign request, the server echoes back the message. However, the attacker's malicious request causes a buffer overflow, because the value of the length field in her request is longer than the content [16]. The result is that the server sends back unintended additional data. The server's response includes the original message plus the data past the end of the message buffer. The exploit is devastating when the data after the buffer is sensitive, like a private encryption key.

While HeartBleed is an example of a confidentiality violation, non-control-data attacks can also result in integrity violations. For example, overwriting the user ID variable with the value of the administrator's user ID is one way to perform a privilege escalation attack. To prevent all non-control-data attacks we need both confidentiality and integrity of sensitive data. Neither of these attacks are caught by any of the mentioned defense mechanisms. Since non-control-data attacks can slip past these defenses and complete memory safety is too costly, we need new tools targeted at providing data protection with low overhead.

3. THREAT MODEL

Our threat model assumes that the attacker can exploit the original benign program to perform arbitrary reads and writes through attacker-controllable memory errors. We assume the system is protected from code injection (or modification) attacks by DEP [52]. We assume that another protection mechanism is in place in the system to prevent control-flow hijacking. Side-channel attacks that leak information from the sensitive region are out of scope. The operating system, our compiler, and our instrumentation are in the trusted computing base (TCB).

4. DCI DESIGN

The key idea behind DCI is that we need protection against non-control-data attacks, but not all data in a program is equally important in terms of security. Relaxing the protection on the non-sensitive memory objects allows us to reduce the overhead of our protection mechanism relative to complete memory safety. Specifically, the non-sensitve checks are more efficient, and no metadata is tracked for non-sensitive objects. DCI ensures that despite the presence of memory vulnerabilities, the confidentiality and integrity of sensitive data is preserved.

Determining the subset of sensitive data is a pivotal deci-

sion. The current DCI implementation uses a programmer specification to describe which data is sensitive. While it would seem ideal to instead use a sophisticated static analysis to automatically infer sensitive data, the programmer intuitively has domain knowledge that is not available to a compiler or a static analyzer.

DCI separates the memory into two regions, a precisely protected region for sensitive data and a coarsely protected region for non-sensitive data. The mechanism enforces this separation and forbids any pointers from the non-sensitive region to the sensitive region. The DCI policy requires sensitive data to be precisely bounds checked in the same manner as complete memory safety. However, the policy relaxes the requirement for non-sensitive data. Pointers to non-sensitive data can be dereferenced if they point anywhere except a sensitive memory object. Such coarse bounds checking for non-sensitive data lowers overhead. Our policy requires coarse bounds checking (sandboxing) non-sensitive data as otherwise, an attacker could access sensitive data through a corrupted non-sensitive pointer. Thus, we must have a check on every pointer dereference, regardless of whether that pointer points to sensitive or non-sensitive data [18].

Finally, data-flow between variables in different regions is forbidden. This policy applies to primitive variables, pointers, and contents of aggregate types. This rule means that an object itself (`struct`, array, etc.) and all its sub-objects (members, elements, etc.) have the same sensitivity. This property is enforced at compile time by separating all variables into one of two disjoint sets – the sensitive set or the non-sensitive set.

Any attempt by the attacker to modify the sensitive data with a non-sensitive pointer, as well as a buffer overflow or use-after-free inside the sensitive region, causes our system to abort the program. An attacker may still modify sandboxed data through non-sensitive pointers, as the security policy does not protect such data.

4.1 Determining the Sensitive Types

DCI requires the programmer to specify which data is sensitive. A recent study showed that annotation burden is a primary factor affecting developers use of contracts [46], which are similar to our annotations. A survey of 21 open source projects that use contracts found that over 33% of program elements (classes or functions) were annotated [17]. For large code bases, annotating functions individually does not scale. To reduce annotation burden, we design our protection specification to be type-based. When the programmer annotates a type, all instances of that type are sensitive. Annotating a type in effect annotates every function that uses that type as a parameter and every local or global variable of that type. This way, a small number of annotations give a specification for the entire program.

For example, to mark all instances of `struct circle` in the entire program as sensitive, the programmer would mark any instance with our provided annotation:

```
__attribute__((annotate("sensitive")))
struct circle c;
```

This has the effect of enabling sensitive protection to:

1. instances of `struct circle` and their contents;
2. instances of other types that contain `struct circle` as a member.

4.1.1 Implicit Sensitivity

A key design decision is how to handle interactions between non-sensitive and sensitive variables. The compiler can reject the program (and display an error message to the programmer) or it could automatically make the variable implicitly sensitive. To minimize annotation burden DataShield defaults to the second option.

Implicit sensitivity potentially leads to more variables being sensitive than the programmer expects, but greatly reduces the number of required annotations. To mitigate the problem of the programmer not knowing exactly what variables are sensitive, the compiler can report a list of sensitive variables and types to which protection was propagated. The programmer can then either iteratively modify the program or the sensitivity specification.

4.2 Sensitivity Rules

This section formalizes the sensitivity rules. To prevent data-flow between sensitive and non-sensitive variables, the first rule forbids direct interaction between variables of different sets.

1. x *[op]* y is only allowed when
   sensitivity(x) = sensitivity(y)

where *[op]* can be any binary operator. Unary operators do not change the sensitivity of their operand.

Constants take on the protection of their other operand:

2. sensitivity(x *[op]* c) $\leftarrow$ sensitivity(x)

where c is any constant.

These rules are sufficient for primitive values, but there are additional rules for pointers:

3. For pointers p and q, if q is *based on* p
   sensitivity(q) $\leftarrow$ sensitivity(p)
4. For pointer p and its pointee *p,
   sensitivity(p) $\leftarrow$ sensitivity(*p)

Here "based on" means that q is the result of pointer arithmetic on p. The additional pointer rules mean that the contents of a `struct` have the same sensitivity as a pointer to the `struct` and array elements have the same sensitivity as the array itself.

The final policy rules are:

5. Sensitive pointers can only be dereferenced within the bounds of a valid sensitive object.
6. Non-sensitive pointers cannot be dereferenced within the bounds of a sensitive object.

Rule 5 prevents overflows between sensitive objects, and as side effect prevents overflows to non-sensitive variables as well (since they are out of bounds of the original sensitive object). It also prevents dereferencing pointers to sensitive objects when they point to unallocated memory (temporal errors). Rule 6 prevents overflows from non-sensitive objects to sensitive objects.

4.3 Enforcement

At runtime, DCI divides memory into two regions, associates metadata with each sensitive pointer in the program, and performs a check before each pointer dereference. A conceptual memory layout is shown in Figure 1.

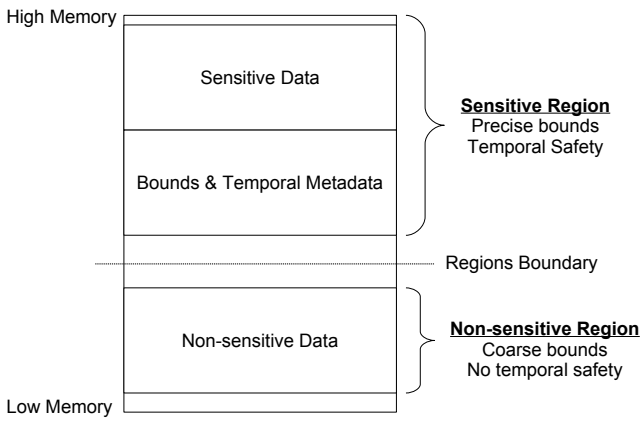

Figure 1: DataShield's runtime memory layout for sensitive and non-sensitive data. The sensitive region has a strict security policy that leads to instrumentation overhead, and the non-sensitive region has a relaxed security policy with minimal overhead.

The non-sensitive region simply contains all the non-sensitive data. The sensitive region holds both sensitive data and the metadata for the sensitive pointers.

Whenever a new sensitive memory object is allocated, the bounds of that object and a temporal check key are recorded as metadata. When a new non-sensitive object is allocated, no metadata is recorded, because non-sensitive pointers have the implicit bounds of anywhere except a sensitive object and no temporal safety. Sensitive and non-sensitive objects are allocated using a region-specific allocator. Pointers in sensitive memory are restricted to point within the bounds of their intended memory object by checking their value against the associated metadata, while non-sensitive pointers may point anywhere in the non-sensitive region. The relaxed policy for non-sensitive pointers leads to more efficient checks and less metadata.

5. DCI IMPLEMENTATION

DataShield implements DCI by extending the LLVM compiler. At a high-level, the compile-time portion of DataShield consists of collecting sensitive types, identifying sensitive variables and inserting new instructions to enforce the DCI policy. The new instructions create metadata, and perform sensitive or non-sensitive bounds checks. The runtime initializes the metadata data structure and implements the checks and region-based allocators. The compiler portion is implemented as an LLVM pass in 4,500 lines of C++ code. The DataShield runtime is 1,000 lines of C code.

5.1 Identifying Annotated Types

Most C/C++ compilers, including GCC and clang/LLVM, already have an annotation facility built in, requiring only minor modifications to support the type annotations that DataShield requires.

Our first pass scans all the code in a module, recording the annotated types as sensitive. When the programmer adds an annotation to her code, it appears in the LLVM IR as metadata. Identifying the set of sensitive types is as simple as parsing the metadata.

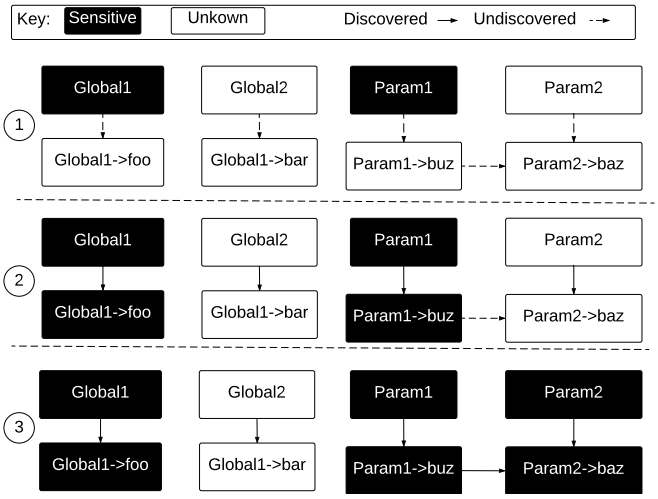

Figure 2: A sensitivity analysis example. In iteration 1, only the sensitivity of globals and of function parameters are known. Then, the analysis applies abstract interpretation over the function body's instructions, discovering new relationships and adding variables into the sensitive set. It concludes when a fixed point is reached in iteration 3. Arrows indicate that connected boxes must be in the same set according to the DCI policy rules.

5.2 Identifying Sensitive Variables

Once our implementation has identified the sensitive types, the compiler locates variables of those types. Declared variables of the sensitive types form the roots of the data-flow graph. The compiler explores every execution path adding new variables to the sensitive set.

The data-flow analysis that finds all the explicitly and implicitly sensitive variables is inter-procedural and context-sensitive. It is a fixed-point analysis where we iteratively add more variables to the sensitive set as shown in Figure 2.

At the start of the analysis, only global variables and function arguments of the sensitive types are in the sensitive set. Variables that have data-flow with other variables in the sensitive set are unioned into the sensitive set. In our formalization, lowercase letters denote variables and uppercase letters denote types. We use the notation $x \in Sens$ to denote the variable x is in the sensitive set, and the notation $sensType(T) = true$ to denote that any of the following are true:

- T is annotated as sensitive;
- pointers to T are annotated as sensitive;
- T is a member of another type U that is annotated as sensitive type;
- T has a member of another type U that is annotated as sensitive type.

Note that the definition is recursive. For instance assume (i) a program has types T, U, and V, and (ii) V is a member of U and U is a member of T. Then if $sensType()$ is true for any of the types, then it is true for all three.

Pointers to primitive types (e.g., `void*`, `int*`, `char*`, or `float*`) are handled specially. We assume that the programmer does not intend to make *all* instances of, e.g., `char*` sensitive. If the programmer annotates a type which

$$\textbf{T x = LoadInst(T* a)}$$
$$\text{if } sensType(T) \vee a \in Sens \vee x \in Sens$$
$$\text{then } Sens \cup \{x, a\}$$
$$\textbf{StoreInst(T x, T* y)}$$
$$\text{if } sensType(T) \vee x \in Sens \vee y \in Sens$$
$$\text{then } Sens \cup \{x, y\}$$
$$\textbf{T x = BitcastInst(U a)}$$
$$\text{if } sensType(T) \vee sensType(U) \vee x \in Sens \vee a \in Sens$$
$$\text{then } Sens \cup \{x, a\}$$

Figure 3: Abstract interpretation transfer function for finding implicitly sensitive variables. Other instructions simply propagate sensitivity.

has a `char*` member, only `char*` based on the (sensitive) parent type are *explicitly* sensitive. Instances of `char*` that are not members of sensitive types are considered non-sensitive initially, but can become *implicitly* sensitive when our analysis discovers data-flow with other sensitive variables. This approach reduces the amount of primitive types (and broadly data at runtime) that need to be sensitive.

The analysis proceeds by abstract interpretation over the function's LLVM IR instructions, applying the transfer function in Figure 3. For brevity, we show only a subset of interesting instructions. The abstract domain for a given variable is whether it belongs to the sensitive set or not. Once a variable belongs to the sensitive set it can never leave the set. The rest of the instructions simply union the sensitivity of their operands. The one exception is LLVM's `CallInst`, as calling a function with mixed sensitivity arguments is legal under our policy. For instance, it could be that the mixed sensitivity arguments do not interact with each other inside the function body. Alternatively, if the mixed sensitivity arguments do interact, the non-sensitive arguments will be promoted to implicitly sensitive when the callee function is analyzed. The `BitCastInst` instruction propagates sensitivity in both directions and never removes sensitivity. If a non-sensitive pointer is cast to sensitive, or a sensitive pointer is cast to non-sensitive, both the original and cast pointers are considered sensitive.

When a callee function with sensitive arguments is discovered by the analysis, we clone a new version of that function which will be reanalyzed and rewritten with the appropriate sensitivity. At the original call site, the call to the original function is replaced with a call to the newly cloned function. The per-call site cloning is crucial because the same function may be called in different contexts with different argument sensitivities. For example, let there be a function with the signature: "`void foo(void* p);`". It is valid to call `foo` with the parameter `p` as any pointer type, and more relevantly to us, with a sensitive or non-sensitive pointer.

When the analysis concludes, each variable is sensitive or non-sensitive. The analysis yields a conservative over-approximation of the sensitive set which, by design, never leads to a security violation. Putting new variables into the sensitive set only enables precise bounds checking for more variables (at potentially increased performance overhead).

6. RUNTIME

At runtime, DataShield separates the sensitive and non-sensitive memory objects by creating two separate memory regions. The non-sensitive region resides in the lower memory addresses up to a fixed address, which is the highest possible non-sensitive address. DataShield uses $2^{32} - 1$ as the end of the non-sensitive region, but this is a configurable parameter. The sensitive memory region resides in the remaining memory addresses above the non-sensitive region.

While the boundary between the regions is fixed, data within the regions need not be stored at any fixed address. This means that our approach remains compatible with randomization techniques (e.g., ASLR).

Sensitive and non-sensitive heap- and stack-allocated variables are moved to the corresponding region. For the current implementation, the non-sensitive region contains a dedicated heap and stack, but there is no sensitive stack. All sensitive stack allocations are rewritten as sensitive heap allocations, so sensitive pointers are only stored in the sensitive heap or in registers. There is nothing inherent in the DCI policy that requires this implementation choice.

In addition to the two memory regions, DataShield needs a data structure for storing bounds and temporal metadata for sensitive memory objects. We store this metadata disjoint from the actual sensitive data to preserve the memory layout. This follows the approach of SoftBound [34], and allows system calls that take sensitive variables to work without modification. A detailed diagram of the memory layout and pointer-to-bounds metadata mapping is shown in Figure 4. Note that the bounds for the non-sensitive objects in Figure 4 are conceptual and so are the absolute addresses shown. Non-sensitive object bounds are not actually stored in the metadata table, they are hard coded in the coarse bounds check instructions. For a thorough discussion of the merits of disjoint metadata please see Nagarakatte et al. [30]. To bounds check each sensitive pointer, we need to store the base and last addresses, meaning we must save 16 bytes for each sensitive pointer. Note from the figure that bounds are created and checked for sub-objects if the type of the sub-object is a sensitive pointer. Sensitive Pointer A and Sensitive Pointer B have their own bounds in this example.

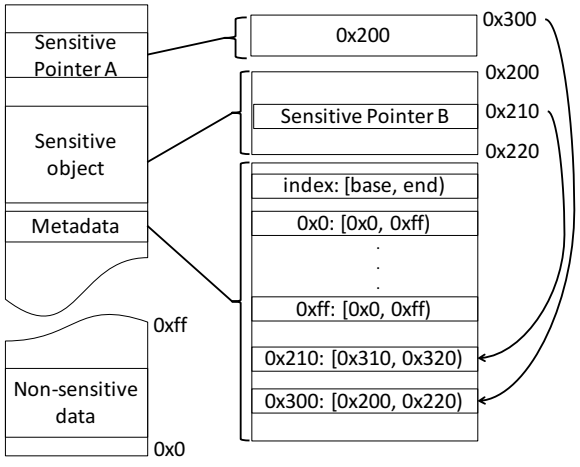

Figure 4: Detailed memory layout, showing the mapping between bounds and sensitive pointers using the pointer's address.

Though unavailable in the current prototype, temporal metadata can be stored in the same metadata table. A discussion on temporal safety is in Section 8.

The runtime must protect the integrity of the metadata table. If the attacker could modify it, she could cause memory errors in the sensitive region. For the current prototype, the metadata table is stored inside the sensitive region. Keeping metadata in the sensitive region allows the coarse bounds check we apply to non-sensitive pointers to protect both the sensitive data and the metadata table.

6.1 Sensitive Globals and Constants

Normally, all constants and global data are loaded together by the program loader. However, to enforce the same security policy for global data as for heap and stack data, we use a linker script to map the sensitive and non-sensitive globals into their respective regions. After the sensitivity analysis finishes, sensitive globals and constants are marked with custom section names which are recognized by our linker script. Any non-sensitive globals are mapped to an address below the sensitive/non-sensitive region boundary. Sensitive globals and constants are instead mapped to an address above the boundary, and metadata is created for them in the same manner as any other sensitive object.

6.2 Instruction Rewriting

The instruction rewriting step occurs after the sensitive variable analysis when the sensitivity of every variable is known. Before every pointer dereference, the compiler inserts the appropriate bounds check depending on the sensitivity of the pointer.

Allocations are replaced with calls to our region-based allocators (based on dlmalloc[1]) that ensure the allocated memory is in the correct sensitive or non-sensitive region.

6.2.1 Rewriting for Non-Sensitive Variables

We have implemented three types of coarse-grained bounds checks. DataShield inserts one of the three following coarse bounds check types, depending on the target processor and configuration, before every non-sensitive pointer dereference. All implementations enforce strong isolation. Considering recent advances in breaking information hiding [18, 42], our prototype avoids information hiding.

Software Mask. The software mask check has the widest compatibility. It only requires the target processor to have an `and` instruction. To mask a non-sensitive pointer, an `and` instruction with a pre-determined value is inserted before the pointer dereference. The mask clears the higher bits of the pointer, preventing the resulting value from pointing into the sensitive region before it is dereferenced.

Intel MPX Bounds Check. Intel MPX adds hardware support for bounds checking, including 4 bounds registers (`bnd0`–`bnd3`) and 7 new instructions. At program startup, our runtime initializes the `bnd0` register with the bounds of the non-sensitive region. The compiler inserts a `bndcu` instruction prior to every non-sensitive pointer dereference. The `bndcu` instruction checks the given pointer value against the upper bound stored in the given bounds register. In our case, it checks the pointer against the bounds of the non-sensitive region. By utilizing the 4 bounds registers, DataShield can support up to 4 non-sensitive regions (e.g., to sandbox different untrusted components) and the non-sensitive regions can reside anywhere in memory.

Address Override Prefix. An address override prefix before an instruction tells the processor to treat address operands as 32-bit values. An instruction with the address override prefix cannot access the sensitive region (since the sensitive region is above 2^{32} in this configuration). This bounds check is supported by any x86-64 processor. On x86-32 processors, previous work used segmentation registers [27], but segments are no longer enforced in x86-64.

6.2.2 Rewriting for Sensitive Variables

Bounds information is created when sensitive memory objects are allocated. The base and last addresses of the allocated object are recorded in the metadata table. Bounds metadata is propagated to other pointers on assignment. For example, extending our `struct circle` example above:

```
struct circle *c1, *c2;

// creates bounds information:
// base = address returned by malloc
// last = base + sizeof(struct circle)*10-1
c1 = malloc(sizeof(struct circle)*10);

// c2 gets assigned the bounds information
// (base and last) from c1
c2 = c1;
```

Prior to every occurrence of a sensitive pointer dereference, the compiler inserts a precise bounds check. The precise bounds check consists of a metadata table look up based on the address of the pointer, and a comparison of the sensitive pointer value with the upper and lower bound retrieved from the table. The coarse bounds enforcement for non-sensitive pointers is much faster than the precise bounds check for sensitive pointers because it consists of at most a single instruction (compared to several instructions and a memory access for the precise bounds check).

6.3 Standard Library Instrumentation

For complete protection, we must instrument both the application itself and the libraries the application uses. DataShield provides instrumented versions of musl[2] for the C standard library, and libc++[3] for the C++ standard library.

For compatibility, DataShield supports shared libraries, as they are used more commonly in practice than statically linked libraries. The issue with shared libraries is that they are compiled separately and ahead of time, without knowledge of the applications the library will be linked against. Since we cannot know all settings the library will be used in, we also cannot know if data-flow in an application will cause a particular library variable to be sensitive. We have two options to address this problem.

Option 1: Two Versions of Each Library. We compile two shared versions of each standard library, one that treats all data as sensitive and another that treats all data as non-sensitive. During compilation of the application, each call to the library is directed to the appropriate version, depending on the sensitivity of the arguments. Note that merging shared state between the two compiled library instances becomes challenging.

Option 2: Drop-in Replacement. We compile a drop-

[1]http://g.oswego.edu/dl/html/malloc.html

[2]https://www.musl-libc.org

[3]http://www.libcxx.llvm.org

in replacement for the default system standard library, i.e., a single shared library that works with programs compiled with and without DataShield's instrumentation. To achieve this, we relocate all library objects to the non-sensitive region and do not insert any checks in the library. Internal checks in the library would fail when linked against applications not compiled with DataShield.

In our evaluation, we use Option 2. Benchmark programs typically make few standard library calls, so checks in libraries should not have a measurable effect on overhead.

Option 2 makes all library data non-sensitive, so application code that deals with non-sensitive data can directly use the library. However, with Option 2, the application cannot pass sensitive data to, or read sensitive data from, the library. Instead, we created wrappers for the standard library functions that propagate metadata (e.g., `memcpy`), and return pointers to library allocated memory objects (e.g., `getenv`.) The functions that return pointers to library allocated objects return pointers to safe region copies. Copies are made during program startup, so there is no opportunity for the attacker to corrupt the sensitive copy. Option 2 offers the added security of checks in the application, while allowing our libraries to be drop-in replacements.

7. PERFORMANCE EVALUATION

To evaluate the efficiency of our implementation prototype, we consider the major contributors to overhead. The first major source of overhead is coarsely bounds checking the non-sensitive object set. The second source is enforcing precise bounds on the sensitive set. There are other sources of overhead, such as initializing and allocating internal data structures, but these happen only once at program start up and are negligible for long lived programs.

A key feature of DCI is that the programmer decides which objects are in the sensitive set. This decision should have an effect on the measured overhead, so our evaluation must account for this decision. This presents a challenge because we cannot evaluate all possible ways to divide the program data into two non-interacting sets. Instead, we perform three experiments, that taken together give an overall picture of DataShield's overhead.

First, we evaluate microbenchmarks designed to vary the split between sensitive and non-sensitive data to quantify the ratio's effect on total measured overhead. We compare DataShield's overhead on these microbenchmarks to SoftBound + CETS, a complete memory safety mechanism, to show the reduced overhead of relaxed protection for non-sensitive data.

Second, we present three case studies where we assumed the role of the programmer. We examined the case study source codes and decided what data should be sensitive. We do not argue that our division of the program data into sensitive/non-sensitive is correct, optimal, or best in an objective sense. In our case studies, we annotated the important data types in the programs, leading to most of the data being sensitive. For our case study programs, we chose libquantum and astar from SPEC CPU2006, and mbedTLS, a TLS/SSL library.

Third, we evaluate the overhead's lower bound, i.e., the sensitive set is empty. This evaluation is an approximation of the case where only a small amount of data in a program is sensitive and it is accessed very infrequently. We evaluate all SPEC CPU2006 C/C++ programs in this configuration.

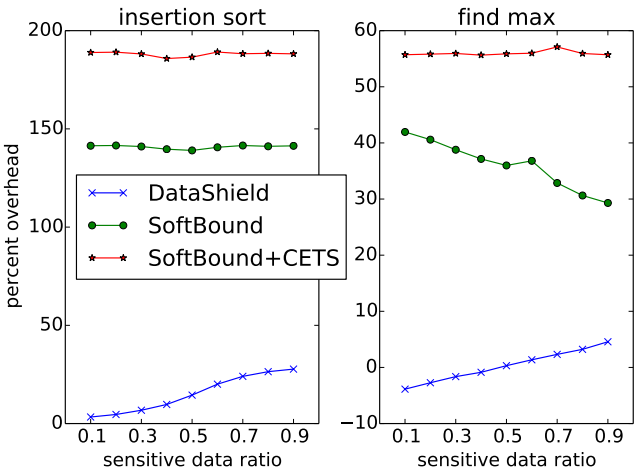

Figure 5: Performance overhead measured on two microbenchmarks when varying the proportion of sensitive to non-sensitive data. More sensitive data leads to higher overhead for DataShield but not for SoftBound + CETS.

For all our evaluations, our platform was Ubuntu 14.04 LTS with an Intel Core i7-6600 3.4 GHz processor and 16 GB of RAM. The baseline compiler was clang 3.9 and all programs were compiled with Link Time Optimization (LTO) and O3 optimizations.

During our evaluation we discovered that our region-based allocator introduced performance speed-ups of up to 20% due to a massive reduction in page faults. We adjusted for this difference by replacing the default allocator with our region-based allocator when measuring baseline performance. Note that we did not modify the allocator used by SoftBound + CETS.

7.1 Microbenchmarks

To quantify the relationship between proportion of sensitive data and overhead, we created two microbenchmarks. In the benchmarks, we create a sensitive and a non-sensitive array, and the size of arrays are varied to control the sensitive to non-sensitive data ratio. We used the software masking implementation of coarse bounds checking for comparison against SoftBound + CETS, because the publicly available implementation uses software bounds checking.

In the first microbenchmark, `insertion-sort`, we sort arrays using insertion sort. This exaggerates the effect of the difference in array sizes because insertion sort's complexity is quadratic. For example, if the non-sensitive array has size N and the sensitive array has size $2N$, then we will execute four times as many sensitive pointer dereferences as non-sensitive.

The second microbenchmark is `find-max`, a simple implementation of a linear scan of an array of objects to find the element with the largest value for a particular integer field. We control the ratio of sensitive to non-sensitive objects by varying the sizes of the two arrays. For example, if the sensitive array has twice as many elements as the non-sensitive array, we know that there should be roughly twice as many sensitive pointer dereferences as non-sensitive – because the `find-max` algorithm is linear in the size of the array.

The results of this experiment are shown in Figure 5.

The overhead of SoftBound + CETS is mostly constant across our experiments, as we would expect. However, as the amount of sensitive data increases, the overhead of DataShield increases towards the SoftBound + CETS overhead. In the figure, SoftBound + CETS includes both spatial and temporal protection, but SoftBound is spatial protection only. All configurations of DataShield, even protecting up to 90% of the data, are faster than SoftBound. Beyond the overhead savings of enforcing memory safety on only a subset of the data, we attribute the additional performance improvements compared to SoftBound to, in part, local optimizations that reduce the number of times bounds are loaded and inlined versions of the checks. We also observed that the region-base allocators can have a large effect on heap locality.

From these experiments, we conclude that non-sensitive data does in fact incur lower overhead using our prototype versus sensitive data. Therefore, the total program overhead is a function of the amount of sensitive data in the program.

7.2 Case Study: libquantum

For our first case study, we evaluated libquantum from the SPEC CPU2006 benchmark suite with a subset of the program's data protected. We decided to protect the `quantum_reg_struct` type as it is one of the main types used by libquantum. To protect this type, we simply added our annotation to the header file that defines the type, i.e., "qureg.h." With precise bounds checking enabled for `quantum_reg_struct` and its sub-objects, we measured an overhead of 27.21% on the `ref` SPEC benchmark inputs. Unfortunately, we cannot compare our overhead to SoftBound as the current SoftBound version does not compile libquantum.

The purpose of the case study is not only to measure the performance overhead, but also to evaluate the difficulty of annotation. For this case study, adding just one annotation for `quantum_reg_struct` protected nearly every pointer in the program sensitive because that data type is used so commonly. We created a dynamic profiler to measure how many dereferenced pointers were sensitive versus non-sensitive. We measured only two non-sensitive pointer dereferences in this configuration. Note that benchmarks are geared towards a single purpose with all data heavily connected. This behavior is therefore expected.

7.3 Case Study: mbed TLS

For our second case study, we applied DataShield to mbedTLS, a SSL/TLS library implemented in about $30,000$ lines of C code. There are two main purposes for this case study:

- To show that the type-based annotation approach is scalable to large programs;
- To measure the overhead a system would incur when using a protected SSL/TLS library in practice.

We annotated the type `ssl_context`, which is the most important type used by the library users. Most functions that are visible to clients take a `ssl_context` as a parameter. The context has fields of many different types: primitives, pointers, arrays, and function pointers. We recompiled the mbedTLS library with the context type annotated and built the included programs `ssl_client2` and `ssl_server2` against our protected library.

With only the type `ssl_context` annotated, we successfully protect all cryptography related memory objects in the client and server. In an example run of the server, 52 non-sensitive pointers were dereferenced compared to over 1.6 million sensitive pointer dereferences. Note that in a production web server, it would have many more non-cryptographic functionalities, so in the other areas of the code there would be more non-sensitive pointer dereferences – which incur lower overhead.

Despite having a high percentage of sensitive objects, we measured the fairly low overhead of 35.7% when exchanging one million messages between the client and server. This is partly due to not incurring instrumentation overhead when performing and waiting for IO, as the client and server communicate with each other over a socket. In practice the SSL/TLS client and server would be running on different machines connected across some network, so the time waiting for IO might be even greater.

In conducting this case study we found the type annotations to be straightforward to use, but we encountered a difficulty with function pointers with sensitive arguments. When the pointed-to function takes an explicitly sensitive type as a parameter there is no problem, and the analysis rewrites the pointed-to function correctly. However, if the caller function invokes a callee function through a pointer with an *implicitly* sensitive argument, the analysis can fail to match the callee and caller correctly and consider the argument as non-sensitive inside the callee. This situation always leads to a false positive policy violation in the callee, which luckily cannot lead to a security vulnerability but aborts the program. To address this problem, we added an annotation that marks the sensitivity of function pointer call sites and address-taken functions. We annotated 50 address-taken functions total for both the `ssl_client2` and `ssl_server2`. Most function pointer invocations do not need annotations, because the analysis usually determines sensitivity correctly if the caller uses the pointer at all – versus allocating a new object, not accessing it, and passing a pointer to the object to the callee.

7.4 Case Study: astar

For our third case study, we use astar, which is a SPEC CPU2006 benchmark. It is a path finding library implemented in 4,285 lines of C++. In this case study, we evaluated the effect of relaxing one of the policy rules on the number of bounds checks and performance. Specifically, we removed rule 1 from Section 4.2 for primitive types only. This relaxation allows sensitive primitive values (`int`, `float`, etc.) to leak information when they are added or subtracted with non-sensitive primitives, but leaves full protection in place for pointers. We refer to this related policy as "separation mode." We annotated the type `statinfot` and used the "rivers.cfg" input configuration.

With the full DCI policy enforced the measured overhead was 96%. In separation mode, the overhead was reduced to 9.12% and the number of sensitive bounds checks was reduced from over $100 * 10^9$ to $160 * 10^3$. Our results show that the full policy is quite strict and results in a large portion of the program data being sensitive. However, if we relax the policy, as in separation mode, we can further control the security versus overhead trade-off.

7.5 SPEC CPU2006 Evaluation

To further evaluate the overhead of DataShield, we recompiled each of the SPEC CPU2006 C/C++ benchmarks

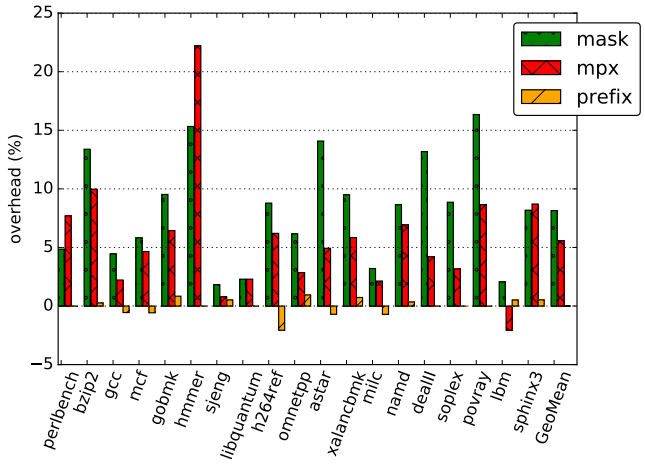

Figure 6: Performance overhead on SPEC CPU2006 for three non-sensitive protection options: masking, Intel MPX, and address override prefix.

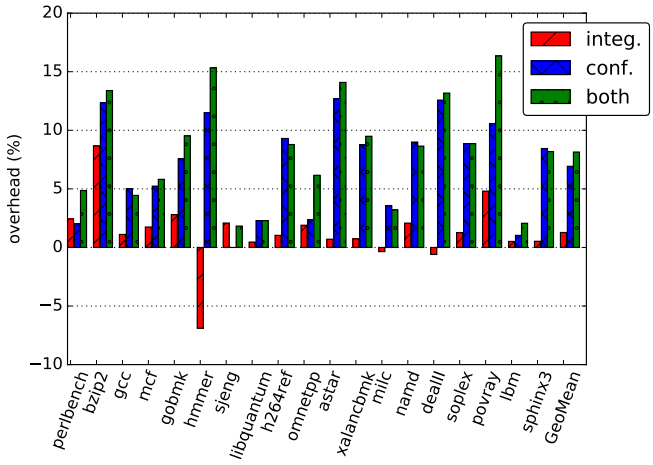

Figure 7: Performance overhead on SPEC CPU2006 isolated by protection type. Integrity-only protects writes, confidentiality-only protects reads, and "both" protects reads and writes.

with our instrumentation. The SPEC benchmarks are not ideal candidates for benchmarking security mechanisms like DataShield. Unlike browsers, webservers, and cryptographic libraries, the SPEC benchmarks are simple programs with few types and none of the benchmarks deal with sensitive data. We include them since they are the de-facto standard for performance measurement.

We did not annotate these benchmarks for this experiment. Even though this experiment is run with an empty sensitive set, the bounds of the non-sensitive region are still enforced. This experiment is effectively measuring the overhead of the parts of a program that do not interact with sensitive data, independent of what sensitive data may exist in the program.

With SPEC CPU2006, we evaluated the three coarse-bounds check options, software mask, Intel MPX, and address override prefix. Moreover, to isolate the components of DataShield's non-sensitive overhead, we measured the overhead of software masking in integrity-only and confidentiality-only modes.

7.5.1 *Comparison of Coarse Bounds Check Implementations*

Depending on the target processor, the programmer may choose among three coarse bounds check implementations, namely software mask, Intel MPX, and address override prefix. Figure 6 shows the overhead of the three options on the SPEC CPU2006 C/C++ benchmarks, using the median of ten runs of each individual benchmark.

For the software mask coarse bounds check, the geometric mean across the benchmarks was 8.14%, and the difference between individual benchmarks is quite large (1.82% to 16.34%). The width of this range is due to some benchmarks having many pointer operations while others having much fewer. For MPX bounds checks and address override prefix the geometric means are 5.56% and 0.0013% respectively.

As expected, the address override prefix implementation had the lowest overhead – too small to measure reliably. The main reason for this is that the address override prefix bounds check does not introduce any additional instructions to the program, it just prefixes existing instructions. The drawbacks are that this is unique to the x86-64 instruction set, and that the prefix applies to a fixed region ($0 - 2^{32}$).

To summarize, using a prefix offers best performance but constrains the location, maximum size of the region, and the ISA. Intel MPX has lower overhead than masking and can give fine-grained control over the location and the size of the region. Masking has the widest compatibility but is the slowest option.

7.5.2 *Integrity and Confidentiality Overhead*

We have evaluated the execution time overhead of DataShield in three different configurations: (i) integrity-only, (ii) confidentiality-only, and (iii) both confidentiality and integrity. These different configurations protect the confidentiality, integrity, or both of the sensitive region.

In integrity-only mode, *only stores* to pointed to memory locations are protected. In confidentiality-only mode, *only loads* from pointed to memory locations are protected. In the third mode, *all loads and stores* are protected.

Integrity-only is clearly useful on its own. Many mechanisms enforce only integrity including CFI, CPI, and WIT [1, 3, 27]. Conversely, confidentiality without integrity is brittle because the attacker can simply overwrite the metadata. We present the overhead of confidentiality-only mode to show the different components of the overhead and for comparison to integrity-only mechanisms. Of course, the enforcement of both integrity and confidentiality is the strongest protection and incurs the highest overhead.

Figure 7 measures the overheads for the different modes on different runs, so integrity-only and confidentiality-only options do not sum up exactly to the combined integrity and confidentiality option due to measurement variation. One interesting aspect of this result is that confidentiality is more costly than integrity, as there are more memory reads than writes in the SPEC CPU2006 benchmarks.

7.6 Security Evaluation

To evaluate the security of our approach, we looked for Common Vulnerabilities and Exposures (CVEs) in our case study programs. Guido Vranken discovered a remote heap corruption vulnerability for mbedTLS in October 2015 [54]

(CVE-2015-5291). The root cause of the vulnerability is a buffer overflow. Specifically, a malicious SSL/TLS server can create a session ticket that overflows the client's buffer when the session ticket is reused by the client, corrupting the client's heap. We recompiled mbedTLS 2.1.1 (an older version, before the vulnerability was patched) both with and without protection, and ran the malicious server against our clients. As expected, without DataShield protection the client's heap was corrupted, but with protection the attack caused a bounds violation and termination of the program. From this evaluation, we conclude that DCI can potentially mitigate vulnerabilities in production software.

Qualitatively, DataShield provides deterministic protection for every sensitive variable in the sensitive set because there is a check on every pointer deference.

7.7 Future Work

In future work, we plan to formalize and improve our sensitivity analysis. As discussed in Section 5.2 we over-approximate the sensitive set. This leads to higher overhead because the security checks on sensitive pointers are more expensive. Therefore, a more precise analysis would result in lower overhead.

Despite not presenting a complete formal proof, we do have some evidence for correctness. If a program runs successfully with instrumentation (which is the case in all our experiments) then we know that every check succeeded. Therefore, the static determination of sensitivity matched the true sensitivity of the pointer at runtime every time a pointer was dereferenced during program execution. This argument, however, does not provide conclusions about the correctness of non-exercised code paths.

We also plan to investigate using Intel MPX to enforce the bounds for both sensitive and non-sensitive pointers at runtime. We would need to change our metadata layout to allow Intel MPX's bounds look up instructions to access it.

8. LIMITATIONS

Our DataShield prototype does not support bounds or temporal checks of variadic arguments (a limitation shared with related work, e.g., SoftBound [34]). This is an engineering issue, because for non-variadic functions we use the function signature to match the function arguments between the caller and callee. However, the variadic function may retrieve the variadic arguments in arbitrarily complicated ways. A straight-forward solution to this problem requires adding an argument to the variadic function prototypes and dynamically reading this new argument when *va_arg* is called to get the variadic arguments. This new argument would specify the number of arguments and the bounds and temporal metadata for each argument at the specific call site. A similar approach was proposed (but not implemented) in SoftBound [34]. In contrast, it is completely safe to pass non-sensitive pointers to variadic functions. We consider all non-sensitive pointers to have the same metadata, so we side-step the problem of matching up arguments to metadata across the caller/callee boundary.

Temporal metadata checking and tracking is not enforced in our current prototype. We could extend our prototype with temporal safety in the same manner that CETS [33] added temporal safety to SoftBound [34]. Following this plan, adding full temporal safety is an engineering effort.

The prototype does provide some temporal protection in that even if a pointer points to deallocated memory, it is impossible for a new object of the wrong sensitivity to be allocated in the pointed-to location. Concretely, given some sensitive pointer P, if $free(P)$ is called, the memory pointed to by P will be available to be reallocated. The most harmful type of temporal error occurs when a new object of a different type is allocated to where P points. However, DataShield mitigates this by guaranteeing that only sensitive objects will be allocated to where P points. The attack surface is limited by requiring a temporal error to exist in the portion of the program that uses sensitive data. Or in the case where there is only one sensitive type, DataShield provides region-based temporal safety analogously to DieHard(er) [5, 41] and Cling [2].

9. RELATED WORK

There are many proposed techniques that aim to add memory safety to C or a C dialect. Approaches that augment the C language include CCured [35] and Cyclone [26]. Both approaches are compiler-based and combine static analysis with runtime checks. DataShield is inspired by CCured and Cyclone in that it tries to make the porting process as easy as possible. There is a massive amount of legacy C code for which porting to a new language is too costly.

SoftBound [34] provides complete spatial memory safety but works on unmodified C code. CETS [33] is an extension to SoftBound that provides temporal safety. The main drawback of SoftBound + CETS, and complete memory safety in general, is overhead. Code-Pointer Integrity (CPI) [27] is a specialization of memory safety that only protects code pointers. This ensures control-flow integrity while reducing overhead relative to complete memory protection. DCI extends CPI's partial protection to other types of data. Key differences between CPI and DCI are:

1. CPI protects code pointers only (e.g., function pointers, return addresses, or indirect jumps) while DCI protects any type of data, not just pointer data;
2. DCI allows the programmer to specify what is protected whereas CPI only focuses on code pointers;
3. DCI protects the content of objects along with pointer values whereas CPI protects pointer values only;
4. DCI enforces both integrity and confidentiality where CPI only enforces integrity.

Yarra [47] is similar in concept to the DCI policy but Yarra focuses on programming language theory while our work targets a practical implementation. Yarra has two modes, whole program and targeted. Whole program mode is complete memory safety with metadata for each memory address. The runtime of gzip from SPEC INT2000 in whole program mode is 6x the baseline. In targeted mode, Yarra uses page protection to lock its protected data whenever unprotected functions are executing. This approach was inspired by Samurai [43] and has great compatibility because it can guarantee the integrity of the protected data even when running completely unknown and untrusted code. The drawback is that the overhead of updating the page permission is far higher than our implementation. Yarra's execution time of gzip in targeted mode is 2x the baseline.

Kenali [49] enforces the integrity of kernel security checks with a form of data-flow integrity. It is similar to DCI in that it attempts to infer the sensitive data from a set of

sensitive data root variables. For Kenali, root variables are the error codes returned by kernel security checks but in DCI the roots can be any data type. The protection enforcement is stronger in DataShield than in Kenali. Kenali relies on information hiding to protect its stack and overflows between sensitive objects are not prevented by Kenali. We believe DCI offers a more flexible approach in that the programmer can control which data is sensitive and it works on a variety of programs whereas Kenali targets only the Linux kernel.

Shreds [10] is a new compartmentalization mechanism for protecting sensitive data. Unlike Shreds which treats all memory inside the shred as sensitive, DCI supports code that mixes sensitive and non-sensitive data. Shreds provides no protection against overflows between sensitive objects. If there is a memory error anywhere within the shred, the attacker can corrupt any memory inside the shred.

Several approaches attempt to reduce the memory overhead of complete memory safety [30]. Hardware support [14, 31, 32] has been shown to reduce overhead. ASAP [55] is a tool that allows the programmer to specify the amount of overhead she is willing to accept then only inserts checks up to that budget. DCI also aims to reduce the performance overhead but never relaxes the policy on sensitive data. SAFECode [15] used static analysis to eliminate checks and its allocation pools are similar to DCI if we consider the sensitivity to be part of a variable's type. METAlloc reduces the cost of metadata look up [22]. PAricheck [57] reduces the cost of pointer arithmetic checks by labeling memory objects and checking if the result points to an object with the same label. Similar approaches that are memory allocator based include Cling, DieHard(er), and Baggy Bounds Checking [2, 4, 5, 41].

The inspiration for DataShield comes from the abundance of work on Control-Flow Integrity [1, 7, 25, 27, 37, 39, 45, 53]. CFI mechanisms are becoming robust and practical, but they do not address non-control-data attacks. Chen et al. [9] argued that non-control-data attacks pose a significant and realistic threat. DataShield's protection scheme is similar to the implementation of Monitor Integrity Protection (MIP) [38] in that both enforce separated regions. The monitors in MIP are analogous to the sensitive data in DCI. However, the main distinction is DataShield enforces both confidentiality and integrity. Other similar isolation mechanisms include PittSFIeld [28] and Native Client [56]. Recent work has called into question the security of CFI. Control-flow Bending [8], Control-flow Jujutsu [19], Counterfeit Object Oriented Programming [48], and Out Of Control [21] showed there are multiple attack vectors to bypass CFI.

10. CONCLUSION

With control-flow protection mechanisms maturing and transitioning to practice, attackers will shift to data-only attacks. Unfortunately, non-control-data attacks are equally devastating as control-flow hijack attacks and we must address this threat. However, existing mechanisms either exhibit prohibitive overhead or cannot detect data-only attacks. To address this issue, we have proposed a new security policy called Data Confidentiality and Integrity (DCI) that selectively protects sensitive data.

We have designed DCI with the security/overhead tradeoff in mind. Precise bounds and temporal checking on *all* memory objects in the program imposes prohibitively high overhead, but allowing the programmer to specify a *subset of sensitive data* to *fully protect* mitigates this issue. We have measured significantly reduced overhead, relative to complete memory safety, on the SPEC CPU2006 benchmarks, our microbenchmarks, and case studies. DataShield's strong protection is evidenced by our security evaluation which showed that it mitigates vulnerabilities in mbedTLS, a production quality SSL/TLS library.

11. ACKNOWLEDGMENTS

This work was sponsored, in part, by NSF grants number CNS-1464155, CNS-1513783, and CNS-1657711 and a gift from Intel. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. CCS 2005.

[2] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. USENIX Security 2010.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *S&P 2008*.

[4] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security 2009*.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. PLDI 2006.

[6] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS '11*.

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Protection, Security, and Performance. In *CSUR*, 2017.

[8] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security 2015*.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data Attacks Are Realistic Threats. SSYM 2005.

[10] Y. Chen, S. Reymondjohnson, Z. sun, and L. Lu. Shreds: Fine-grained Execution Units with Private Memory. In *S&P 2016*.

[11] P. Collingbourne. LLVM — Control Flow Integrity, 2015. http://clang.llvm.org/docs/ControlFlowIntegrity.html.

[12] 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*.

[13] 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. SSYM 1998.

[14] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. ASPLOS XIII (2008).

[15] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode:

Enforcing Alias Analysis for Weakly Typed Languages. PLDI 2006.

[16] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The Matter of Heartbleed. In *IMC 2014*.

[17] H.-C. Estler, C. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in Practice. In *FM 2014: Formal Methods*.

[18] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point: On the Effectiveness of Code Pointer Integrity. In *S&P 2015*.

[19] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *CCS 2015*.

[20] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of fine-grained Control Flow Integrity. 2015.

[21] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *S&P 2014*.

[22] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening. EuroSec 2006.

[23] M. Hicks. What is memory safety. http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

[24] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, May 2016.

[25] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *NDSS 2014*.

[26] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. ATEC 2002.

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *OSDI 2014*.

[28] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security 2006*.

[29] Microsoft Corporation. Control Flow Guard (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[30] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *SNAPL 2015*.

[31] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. ISCA 2012.

[32] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. CGO 2014.

[33] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. ISMM 2010.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. PLDI 2009.

[35] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*

[36] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):http://phrack.com/issues.html?issue=67&id=8, Nov. 2007.

[37] B. Niu and G. Tan. Modular Control-flow Integrity. PLDI 2014.

[38] B. Niu and G. Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. CCS 2013.

[39] B. Niu and G. Tan. Per-Input Control-Flow Integrity. CCS 2015.

[40] B. Niu and G. Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. CCS 2014.

[41] G. Novark and E. D. Berger. DieHarder: Securing the Heap. CCS 2010.

[42] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking Holes in Information Hiding. In *USENIX Security 2016)*.

[43] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting Critical Data in Unsafe Languages. Eurosys 2008.

[44] PaX-Team. PaX ASLR. http://pax.grsecurity.net/docs/aslr.txt, 2003.

[45] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *DIMVA 2015*.

[46] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case Studies and Tools for Contract Specifications. ICSE 2014.

[47] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Modular Protections against Non-Control Data Attacks. In *CSF 2011*.

[48] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P 2015*.

[49] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *NDSS 2016*.

[50] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. S&P 2013.

[51] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security 2014*.

[52] A. van de Ven and I. Molnar. Exec Shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[53] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. CCS 2015.

[54] G. Vranken. CVE-2015-5291: remote heap corruption in ARM mbed TLS / PolarSSL, October 2015.

[55] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High System-Code Security with Low Overhead. In *S&P 2015*.

[56] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P 2009*.

[57] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. ASIACCS 2010.