DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection

Tapti Palit
Stony Brook University
tpalit@cs.stonybrook.edu

Jarin Firose Moon
Stony Brook University
jfmoon@cs.stonybrook.edu

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—As control flow hijacking attacks become more challenging due to the deployment of various exploit mitigation technologies, the leakage of sensitive process data through the exploitation of memory disclosure vulnerabilities is becoming an increasingly important threat. To make matters worse, recently introduced transient execution attacks provide a new avenue for leaking confidential process data. As a response, various approaches for selectively protecting subsets of critical in-memory data have been proposed, which though either require a significant code refactoring effort, or do not scale for large applications.**

In this paper we present DynPTA, a selective data protection approach that combines static analysis with *scoped* dynamic data flow tracking (DFT) to keep a subset of manually annotated sensitive data always encrypted in memory. DynPTA ameliorates the inherent overapproximation of pointer analysis—a significant challenge that has prevented previous approaches from supporting large applications—by relying on lightweight label lookups to determine if *potentially* sensitive data is *actually* sensitive. Labeled objects are tracked only within the subset of value flows that may carry potentially sensitive data, requiring only a fraction of the program's code to be instrumented for DFT. We experimentally evaluated DynPTA with real-world applications and demonstrate that it can prevent memory disclosure (Heartbleed) and transient execution (Spectre) attacks from leaking the protected data, while incurring a modest runtime overhead of up to 19.2% when protecting the private TLS key of Nginx with OpenSSL.

I. Introduction

As defenses against control flow hijacking attacks become more widely deployed, attackers have started turning their attention into data-only attacks [1] for the exploitation of memory corruption or disclosure vulnerabilities. Under certain conditions, the corruption of non-control data can lead to arbitrary code execution, e.g., by re-enabling the execution of untrusted plugins [2, 3, 4]. As technologies such as Flash and ActiveX are being phased out, mere *data leakage* is still possible and can pose a significant threat, e.g., the exfiltration of secret server keys [5] or private user information [6]. As if the abundance of memory disclosure bugs was not enough, the threat of data leakage attacks has recently been exacerbated by a flurry of transient execution attacks [7], which can leak secrets through residual microarchitectural side effects. Examples of the severe outcomes of these attacks include accessing security-critical files, such as `/etc/shadow` [8, 9], and leaking memory from Chrome's renderer process [10].

Various defenses can be used against data leakage attacks, involving different performance, usability, and completeness tradeoffs. Holistic approaches against memory corruption bugs, such as memory safety [11, 12, 13, 14, 15], and data flow integrity [16], can mitigate data leakage attacks by eradicating their main exploitation primitive, i.e., arbitrary memory read access. In practice, however, their deployment has been limited due to their prohibitively high runtime overhead and incompatibility with C/C++ intricacies that are widely used in real-world applications [17]. At the same time, they inherently cannot protect against transient execution attacks, many of which focus on precisely bypassing such software-enforced bounds checking and similar policies [18, 19].

Instead of protecting all data, an alternative approach is to *selectively* protect only the subset of data that is really critical for a given program. This can be achieved in several ways, including privilege separation [20, 21], secure execution environments [22, 23], sandboxing [24, 25, 26, 27], and fine-grained memory isolation [27, 28, 29, 30]. Although these approaches differ across various aspects, their common characteristic is that they all require a significant code refactoring effort, which is particularly challenging for large applications.

Seeking to increase the practical applicability of selective data protection, some recent approaches have opted for requiring the programmer to just *annotate* security-critical memory objects in the source code as "sensitive," and then automatically harden the program to keep this data protected [31, 32, 33]. This is achieved by a compiler pass that identifies and instruments the memory load and store instructions that operate on sensitive objects. DataShield [31] inserts fine-grained bounds checks for pointers to sensitive data, and lightweight coarse-grained bounds checks for other pointers. Glamdring [32] inserts transitions to and from an Intel SGX [34] enclave that holds the sensitive data. Selective in-memory encryption [33] inserts cryptographic transformations to keep the in-memory representation of sensitive data always encrypted.

A key component of these approaches is the automated identification of all instructions that may access sensitive memory locations. Due to the widespread use of pointers in C/C++, pointer (or *points-to*) analysis must be used to resolve which pointers can point to sensitive memory locations. There has been extensive research in the area of points-to analysis, with various algorithms falling at different points in the spectrum of precision vs. speed. Andersen's algorithm [35] offers increased precision, but with a computational complexity of $O(n^3)$ that makes it inapplicable to large programs. Indicatively, based on our experience with SVF's [36] Andersen's implementation, it

takes about 11 hours to complete for Nginx with OpenSSL, while for the Chromium browser, our machine (with 32 GB of RAM) ran out of memory after running for four days. On the other hand, Steensgaard's algorithm [37] has an almost linear time complexity of O(n), making it scalable for large programs, but comes with a much higher level of imprecision.

The issues of scalability and precision in points-to analysis are well-known (we refer to Hind et al. [38, 39] for a more detailed discussion). For selective data protection defenses, the more imprecise the points-to analysis, the higher the number of memory operations identified as *potentially* sensitive—due to the overapproximation in computing the points-to graph, more memory locations than those that will actually hold sensitive data must be protected. This in turn requires more memory instructions to be instrumented, which leads to higher overhead. This challenge limits the applicability of prior works [31, 32, 33] to only moderately complex programs, such as MbedTLS (a TLS library tailored for embedded systems). To the best of our knowledge, no prior work on selective data protection is applicable to larger code bases, such as OpenSSL, which has actually suffered from in-the-wild data leakage attacks [5].

The competing requirements of high precision (to reduce instrumentation and its overhead) and reasonable computational complexity (to scale the analysis for large programs) motivated us to rethink our approach towards pointer analysis. Starting with the goal of making selective data protection practical for large applications, in this paper we present *DynPTA*, a defense against data leakage attacks that combines static analysis with dynamic data flow tracking (DFT) to keep a subset of manually annotated sensitive data always encrypted in memory. To protect sensitive data from leakage, we opted for in-memory encryption [33] because i) it makes the overall approach applicable on a wide range of systems (in contrast to relying on more specialized hardware features [30, 32]), and ii) it protects against transient execution attacks, as any leaked data will still be encrypted (in contrast to memory safety [31]).

DynPTA uses the linear-time Steensgaard's points-to analysis to support large programs, but ameliorates the overapproximation of the computed points-to graph by relying on lightweight label lookups to determine if *potentially sensitive* data is actually sensitive. We introduce a *scoped* form of dynamic DFT to track labeled objects that is applied only on the potentially sensitive value flows that were identified during static analysis—requiring only a fraction of the program's code to be instrumented for DFT. For a given sensitive pointer dereference, DynPTA selectively encrypts or decrypts the accessed data depending on the presence or absence of the sensitive label. To reduce the imprecision of the points-to analysis even further, we also introduce a summarization-based context-sensitive analysis of heap allocations that results in improved runtime performance.

We implemented a prototype of DynPTA on top of LLVM, and successfully applied it on eight popular applications, including Nginx with OpenSSL, Apache Httpd, and OpenVPN—applications with an order of magnitude more lines of code compared to programs such as MbedTLS that were used in previous works [31, 33]. DynPTA incurs a modest runtime overhead of up to 19.2% when protecting the private TLS key of Nginx with OpenSSL, while for MbedTLS the overhead is just 4.1% (in contrast to a reported 13% for in-memory encryption [33] and 35% for DataShield [31]). We also evaluated DynPTA with real-world memory disclosure (Heartbleed [5]) and transient execution (Spectre-PHT [40] and Spectre-BTB [41]) attacks, and demonstrate that the protected data always remains safe.

In summary, we make the following main contributions:

- We propose a hybrid approach that combines static analysis with scoped dynamic data flow tracking to improve the scalability and accuracy of points-to analysis.
- We propose a summarization-based context-sensitive heap modeling approach that reduces the overapproximation of points-to analysis for heap allocations.
- We implemented the above approaches in DynPTA, a compiler-level selective data protection defense that keeps programmer-annotated data always encrypted in memory.
- We experimentally evaluated DynPTA with real-world applications and demonstrate that it can protect against memory disclosure and transient execution attacks while incurring a modest runtime overhead.

Our implementation of DynPTA is publicly available as an open-source project at https://github.com/taptipalit/dynpta.

II. Background and Motivation

A. Pointer Analysis

Static *pointer analysis* computes the potential targets of pointers in a program. Pointer analysis is sound, but as a *static* analysis technique, it lacks access to critical runtime information, and therefore suffers from overapproximation, i.e., the resulting "points-to" set of a pointer may include objects that the pointer will never point to at runtime.

Pointer analysis assumes that a pointer may only point within the valid bounds of the target object. Memory disclosure vulnerabilities can still dereference a pointer to access memory beyond these bounds and leak other objects. Pointer analysis can correctly identify and model other classes of pointer transformations that are considered undefined by the ANSI C standard, such as casting an integer value to a pointer. These undefined transformations, however, result in major loss of precision in the resulting points-to graph.

1) Set Inclusion vs. Set Unification: Andersen's [35] inclusion-based and Steensgaard's [37] unification-based algorithms are the two most common pointer analysis approaches. Both begin by iterating over every instruction and collecting *constraints* related to the flows of pointers. These constraints are of the types *Addr-of*, *Copy*, *Deref*, and *Assign*. For C programs, these correspond to statements of the form $p := \&q$, $p := q$, $p := *q$, and $*p := q$, respectively. Each algorithm solves these constraints using a set of *resolution rules*.

Andersen's analysis begins by constructing *points-to sets* for each pointer. When a new possible target `q` is found for a pointer `p`, then `q` is *included* in the points-to set of `p`. This inclusion, however, requires the recomputation of the pointer

relationships for all Deref constraints that involve p, resulting in a cubic complexity of $O(n^3)$. This makes Andersen's algorithm inapplicable to large and complex applications.

Steensgaard's analysis maintains both *pointer sets* and *points-to sets*. Every pointer is a member of a unique set, and a points-to relationship is represented as a one-to-one mapping between a pointer and a points-to set, i.e., *all* pointers in a pointer set *may* point to *all* objects in a points-to set. When a new target q is found for a pointer p, p's points-to set is *unified* with the set that contains q. This allows the algorithm to run in almost linear time, making it applicable to large applications. However, the unification of pointer sets leads to a significant loss of precision, which makes the analysis results less useful.

We discuss the constraints and resolution rules in more detail and also illustrate the results of running both Andersen's and Steensgaard's analysis on a C code snippet in Appendix A.

2) Memory Object Modeling: Although constraint resolution is an important consideration, the way constraints are modeled also affects the analysis precision and speed. *Context sensitivity* is a constraint modeling approach that considers the calling context when analyzing the target of a function call. When a function is invoked from multiple call sites, each site is analyzed independently, reducing imprecision. This is critical for functions that allocate or reassign objects referenced by their arguments, or functions that return pointers. The prevalent use of wrapper functions around memory allocation routines makes context sensitivity a particularly important issue. Performing context-insensitive analysis for memory wrappers would cause *all* pointers to heap memory to point to the same object. We address this issue by introducing a summarization-based context-sensitive heap modeling approach (Section IV-B).

B. In-memory Data Encryption

Starting with the programmer's sensitive data annotations, the results of value flow analysis and pointer analysis provide us the set of all memory load and store instructions that may access sensitive memory locations. How these memory locations will be protected against data leakage attacks is an orthogonal design decision with various possible options. Relying on typical software-based memory safety checks [31] requires the insertion of just a few instructions per memory access, but does not offer any protection against transient execution attacks [7, 19]. Relying on hardware-enforced memory isolation [30, 32, 42] can potentially reduce the cost of memory protection, but the coarse-grained nature of these isolation mechanisms make them challenging to use for individual memory objects, while they may not be available on legacy systems.

An alternative is to keep sensitive data always encrypted in memory, and decrypt it only when being loaded into CPU registers [33]. Leaking secrets from registers requires arbitrary code execution, which falls outside our threat model. The main benefits of this approach include protection against transient execution attacks, and wide applicability on existing and legacy systems. The main drawback is the exceedingly high runtime overhead of cryptographic transformations, even with hardware acceleration through the AES-NI extensions.

For DynPTA, we opted to protect sensitive data using in-memory encryption due to its attractive benefits. As the key advantage of our approach is that it ameliorates the overapproximation of the points-to analysis using runtime information, we can afford the cost of cryptographic operations, as they will be applied sparingly.

III. Threat Model

We consider memory disclosure or data leakage vulnerabilities that allow an attacker to read arbitrary user-space memory. Data modification (e.g., swapping an encrypted value with another leaked encrypted value) or corruption attacks are out of the scope of this work. We assume that either due to the nature of the vulnerability (e.g., Heartbleed), or due to defenses and mitigations against arbitrary code execution, the attacker has to resort to a data leakage attack. Given that attackers cannot execute arbitrary machine code, any sensitive information or secrets stored in the processor's registers remain safe. Note that attackers may still run arbitrary *script* code (e.g., in-browser JavaScript) to access any part of the process's address space through a memory disclosure vulnerability [6].

Our focus is on user-space applications, and the exploitation of kernel vulnerabilities is out of scope, as we assume that the attacker cannot corrupt any kernel code or data.

Transient execution attacks can be classified as Spectre-type [18] or Meltdown-type [43], depending on whether the program can access the compromised data architecturally [7]. Spectre-type attacks bypass software-defined security policies, such as bounds checking. Meltdown-type attacks bypass architectural isolation barriers, and allow access to sensitive data using instructions that cause hardware faults. We consider both user-space Spectre-type and Meltdown-type attacks in our threat model, but their kernel variants are out of scope.

Potential *implicit* leakage of sensitive data that takes part in computation that observably affects control flow (e.g., through execution timing side channels) is outside our threat model.

IV. Design

The main goal of DynPTA is to protect sensitive memory-resident process data from leakage. Due to the presence of pointers in C/C++, pointer analysis is required to resolve sensitive memory accesses. DynPTA ameliorates the imprecision of existing (scalable) pointer analysis algorithms by coupling static pointer analysis with dynamic data flow tracking (DFT). In particular, DynPTA uses a *scoped* form of dynamic DFT that maintains labels for only *potentially* sensitive memory objects. For a given sensitive pointer dereference, DynPTA selectively encrypts or decrypts the accessed data depending on the presence or absence of the sensitive label. Although scoped DFT does not improve the precision of pointer analysis per se, it ensures that only sensitive data undergoes expensive cryptographic transformations. The sensitive data to be protected is identified by the developer, who annotates the respective variables or pointer initialization locations in the source code—no further manual code modifications are required, and the rest of the process is fully automated.

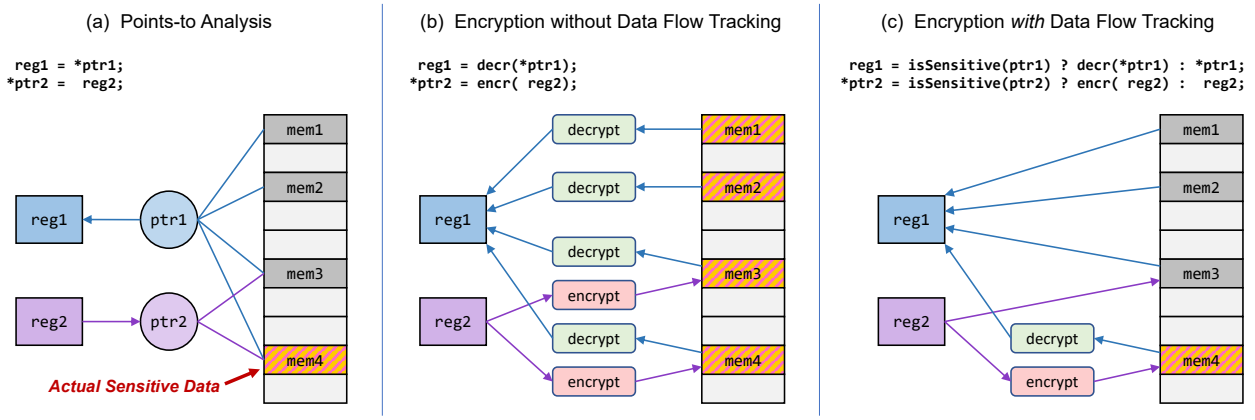

Fig. 1: In this example, $mem4$ has been marked as sensitive, and it can be accessed through $ptr1$ and $ptr2$, along with other memory locations (a). Relying on points-to analysis alone necessitates treating all target locations as sensitive, resulting in excessive cryptographic transformations (b). Before dereferencing a pointer, DynPTA relies on scoped dynamic data flow tracking to first check if the object is truly sensitive, and only then performs the required encryption or decryption operation (c).

Without the use of data flow tracking, the inherent overapproximation of pointer analysis would result in an excessive number of cryptographic operations for data that is not actually sensitive. Figure 1 illustrates how the use of scoped DFT dramatically reduces the required instrumentation, by protecting only the data that is actually sensitive. Consider the sample code snippet in Figure 1(a). In this example, we assume the programmer has specified that location $mem4$ contains sensitive data. The two memory load and store instructions are performed via pointers, and the pointer analysis algorithm resolves $pts(ptr1) := \{mem1, mem2, mem3, mem4\}$, and $pts(ptr2) := \{mem3, mem4\}$. These results may contain overapproximation, i.e., the memory locations that will be accessed through the two pointers may be fewer than the locations the points-to analysis denotes as potential targets.

As shown in Figure 1(b), relying solely on static analysis, we would conclude that $ptr2$ may point to sensitive memory $mem4$, and therefore the value being stored must be encrypted first. As $ptr2$ may also be used to store values to $mem3$, the content of $mem3$ will end up being encrypted as well. Similarly, the pointer analysis informs us that $ptr1$ may be used to read not only from $mem3$ and $mem4$, but also from two more memory locations. Since any read access through $ptr1$ will first decrypt the fetched data, *all* memory objects that $ptr1$ may point to ($mem1$, $mem2$, $mem3$, $mem4$) must be kept encrypted in memory to maintain the correct execution of the program—otherwise any non-encrypted data accessed through the pointer would be mangled by the decryption operation.

Instead of unconditionally encrypting (or decrypting) all memory objects written (or read) through a pointer associated with sensitive data, DynPTA uses *scoped* dynamic DFT to maintain labels for sensitive objects. At runtime, DynPTA *selectively* applies cryptographic transformations depending on the presence or absence of the sensitive label for a given pointer dereference. As shown in Figure 1(c), before reading through $ptr1$ or writing through $ptr2$, DynPTA first dynamically checks whether the pointed object is truly sensitive, and if so, then applies the necessary decryption or encryption operation.

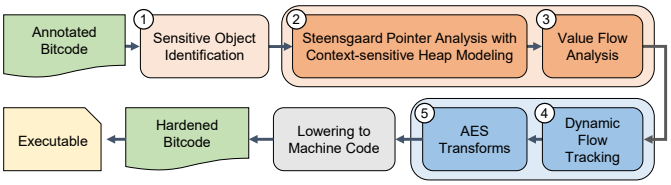

Fig. 2: DynPTA's main analysis and transformation phases.

Figure 2 presents an overview of DynPTA's design, and illustrates how the different phases of our approach analyze and transform a target program. Based on the programmer's annotations, we first identify the initial set of sensitive memory objects ①. The whole code is then analyzed using Steensgaard's algorithm in conjunction with our context-sensitive heap modeling (Section IV-B) to identify the set of memory instructions that may access sensitive data ②. Memory instructions are then further analyzed to pinpoint those that may result in the flow of sensitive values from the initial (annotated as) sensitive objects to other variables ③. Finally, the memory instructions identified in the previous step are instrumented with code that i) determines at runtime whether the read (or written) data is sensitive or not based on its DFT information ④, and ii) in case it is sensitive, decrypts (or encrypts) the data before moving it to CPU registers (or writing it back to memory) ⑤.

A. Sensitive Object Identification

As shown in the example of Listing 1, DynPTA provides a `mark_sensitive()` function that programmers can use to mark individual objects that need to be protected. The function treats the object whose address is provided to it as sensitive. These objects can be simple variables or data referred to by pointers. Note that when a programmer marks a pointer as

sensitive, their intent is to guarantee the confidentiality of *what* the pointer *points to*, and not of the pointer itself.

For pointers, `mark_sensitive()` must be applied at every initialization (or reinitialization) point of the pointer, where the pointer points to a new object. In the example of Listing 1, the `priv_key` pointer is annotated as sensitive after it is initialized in Line 3, and again after it is reinitialized in Line 12. For variables, the programmer must use `mark_sensitive()` only once after the variable is defined.

```
1  int main (void) {
2      char* priv_key = malloc(8);
3      mark_sensitive(priv_key);
4      ...
5      char* ptr = priv_key;
6      ...
7      pub_key[i] = *(ptr+i)^0xA;
8      ...
9      priv_key = malloc(8);
10     mark_sensitive(priv_key);
11 }
```

Listing 1: Simplified code with a pointer annotated as sensitive.

Once the initial annotations are provided by the programmer, no other manual intervention is required. DynPTA then processes the annotations to identify all sensitive objects and applies a "sensitive" label to them that is propagated by DFT at runtime. Identifying sensitive variables is straightforward based on the accompanying annotation. In case a pointer is marked as sensitive, we have to treat the objects that this pointer points to as sensitive, and any memory instructions operating on these objects must be protected. DynPTA uses Steensgaard's pointer analysis with a novel context-sensitive heap modeling approach to find these memory instructions.

B. Summarization-based Context-sensitive Heap Modeling

As discussed in Section II-A2, context sensitivity is an important aspect of modeling the memory of a program for pointer analysis. Most pointer analysis implementations model every call to known Libc memory allocation routines uniquely. For example, in the assignment `p = malloc(...)`, the object allocated by `malloc` flows to the pointer `p`. In the presence of memory allocation wrapper functions, however, this modeling results in a completely *context-insensitive* heap. Although the object returned by the Libc function within the wrapper flows to a single pointer, that pointer itself flows to *all* the call sites that invoke the memory allocation wrapper.

Figure 3(a) shows how existing pointer analysis algorithms model a simplified code snippet from OpenSSL, in which the `CRYPTO_malloc` wrapper is used to allocate memory for the session (`sess`) and certificate (`cert`) objects. The context insensitivity due to the use of the wrapper causes overapproximation, and both `sess` and `cert` point to the same heap object. In practice, the overapproximation is much worse, as *all* memory allocations in the library are performed via calls to `CRYPTO_malloc`, and thus *all* pointers to heap objects would end up pointing to the same object. For our purposes, even if just one of these pointers is marked as sensitive, then all heap objects would become sensitive. To deal with these

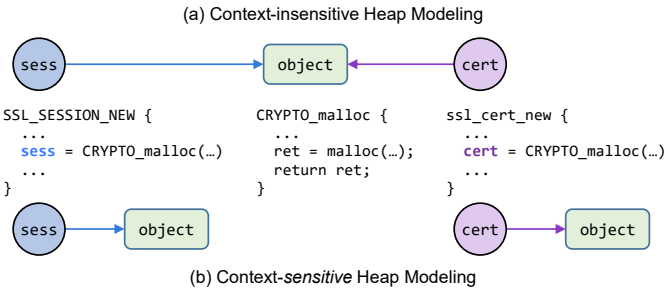

Fig. 3: Context-insensitive (a) vs. context-sensitive (b) modeling of OpenSSL's heap. Without context sensitivity, pointer analysis assumes that `sess` and `cert` point to the same object (a). Using summarization for `CRYPTO_malloc`, overapproximation is reduced by creating two distinct heap objects at its two call sites, allowing pointer analysis to distinguish that `sess` and `cert` point to different objects (b).

challenges, we have developed a summarization-based context-sensitive heap modeling approach tailored to the extensive use of memory-related wrappers in popular applications.

1) Memory Allocation Wrapper Identification: The first step in modeling a context-sensitive heap is to identify the memory allocation wrappers used by a given application. A wrapper typically allocates heap memory via a standard Libc memory allocation function, such as `malloc`, performs some additional sanitization and checks, and returns the pointer to the allocated memory. This pointer, however, may not be the *same* one returned by the Libc function—that pointer may have been copied to other pointers, one of which in turn may be returned. Similarly, in case of pool-based allocators, the memory is allocated in pools and the wrapper returns a pointer into a chunk within this pool. To track such potential pointer manipulation, we perform a lightweight intraprocedural pointer analysis only on the candidate function under consideration, and identify if the returned pointer *always* points to the heap memory allocated via known memory allocation functions provided by Libc.

Another challenge is that memory allocation wrappers may be nested. For example, in OpenSSL, `CRYPTO_malloc` internally invokes Libc's `malloc`, but there are other wrappers around `CRYPTO_malloc`, such as `CRYPTO_remalloc` and `CRYPTO_realloc`, which also need to be identified. Therefore, we begin our analysis with the known Libc memory allocation wrappers from the previous step, but also repeat the process of identifying memory wrappers iteratively, a configurable number of times (currently set to five), with each iteration including the wrappers found in the previous iterations as known memory allocation wrappers.

2) Memory Allocation Wrapper Summarization: The typical way of modeling a context-sensitive memory model for pointer analysis is to reanalyze each function at each call site. In our case, to ensure context-sensitive heap modeling, we would have to reanalyze each memory allocation wrapper at each of their call sites. This comes at a cost of increased analysis time, especially when dealing with nested wrappers.

An alternative, faster approach is to use summarization [44, 45]. Summarization-based approaches analyze each function exactly once to derive the points-to relationships between the arguments and the return values of the function. The result of this analysis is called the *summary* of the function. When performing pointer analysis on the entire program, at each call site of a given function, its pregenerated summary is readily used, instead of analyzing the function again.

We employ summarization by first analyzing each memory allocation wrapper intraprocedurally, and deriving the points-to relationships between its arguments and return values. We store these results in a summary that includes the information that the memory allocation wrapper should allocate a new object on the heap and return a reference to it. As shown in the example of Figure 3(b), our analysis summarizes `CRYPTO_malloc`, and then at each of its call sites, instead of analyzing the wrapper again, its pregenerated summary is used. When the pointer analysis algorithm analyzes these call sites, it creates two different heap objects for the two invocations, and stores separate references to them in the `sess` and `cert` pointers.

C. Pointer and Value Flow Analysis

Once we have modeled the heap allocations in a context-sensitive manner, we analyze all pointers and memory objects in the program using Steensgaard's unification-based pointer analysis algorithm [37]. Every instruction in the program is first analyzed and constraints corresponding to that instruction are collected. Once all constraints are collected, they are solved according to the Steensgaard's algorithm's constraint resolution rules specified in Appendix B, providing us the final points-to sets for each pointer in the program.

Resolving all pointer references is not enough to achieve complete data protection, as sensitive data may propagate to other variables and objects, which we call *sensitive sink sites*. To prevent potential information leakage through these variables, DynPTA performs static value flow analysis to identify all sensitive sink sites.

Sensitive values might flow through both direct and indirect (via pointers) memory instructions, and thus DynPTA tracks both *direct* and *indirect* value flows. Value flows are represented as directed dependency chains originating at a *memory load* operation and terminating at a *memory store* operation. A *sensitive* value flow originates at a load operation from a sensitive memory location, and results in marking the destination memory operand of the final memory store as sensitive. All such directed dependency chains are linked recursively, until no new chain is found.

To track indirect value flows, we use the results of Steensgaard's analysis, and consequently, the value flow analysis provides a *superset* of all value flows that *may* result in the flow of sensitive values. Because the sources of these indirect value flows may include memory loads via pointers, and similarly the destinations of these value flows may include memory stores via pointers, this superset has imprecision associated with both the sources and the destinations of the value flows. These source and destination pointers may point to both sensitive and non-sensitive memory objects. Consequently, if all objects discovered through DynPTA's static analysis were marked as sensitive, we would be unnecessarily protecting a severely overapproximated set of objects.

The actual sources and targets of the identified (potentially sensitive) indirect memory accesses are available at runtime—at which point it can be determined if they are indeed sensitive or not. Below, we describe how DynPTA uses runtime information in the form of labels maintained by scoped dynamic data flow tracking to mitigate the overapproximation of the static analysis.

D. Scoped Dynamic Data Flow Tracking

The result of Steensgaard's algorithm is the superset of *all* possible memory accesses that *may* read from or write to sensitive memory locations. Due to the inherent overapproximation of points-to analysis, this set may include indirect memory operations that actually do not access any sensitive object, as well as indirect memory operations through *partially sensitive pointers*, which access sensitive data only during some of their invocations. Similarly, value flow analysis captures all value flows that may involve sensitive data. At runtime, however, only a subset of them will actually involve sensitive data.

To deal with these two cases of overapproximation, we use *scoped* byte-level dynamic data flow tracking, which relies on a shadow memory to associate labels to the tracked memory locations. Labels are initialized for every object that is marked as sensitive. Then, dynamic DFT is applied only within the scope of the identified *potentially sensitive* value flows, and thus only a fraction of the whole program's code has to be instrumented with DFT propagation logic. The (propagated) sensitivity labels are then used to perform lightweight lookups when dereferencing partially sensitive pointers, to decide whether the accessed object must undergo cryptographic transformations.

1) Dynamic DFT on Potentially Sensitive Value Flows: Every load–store dependency chain identified as potentially sensitive by the value flow analysis (Section IV-C) consists of at least two instructions—a memory load and a memory store. If a dependency chain involves an indirect memory access (via a pointer), then DynPTA instruments all instructions in the chain with DFT logic to propagate label information. As we show in Section VI-B, only a fraction of all value flows (1–9%) end up being instrumented with DFT propagation logic. At the terminating memory store operation, DynPTA determines at runtime whether the value being stored is sensitive (that is, if it was loaded from a sensitive memory location) or not.

If the initial load instruction reads from a sensitive memory location, DynPTA performs two actions: i) it applies the sensitive label to the destination operand of the store instruction, and ii) it encrypts the value being stored so that the in-memory representation of the value is protected against data-leakage attacks. Similarly, if the memory store operation performs an indirect memory access and writes to a memory location via a pointer, the sensitive label is applied only to the memory object that the pointer points to at runtime. Because we include all targets of sensitive value flows identified statically when

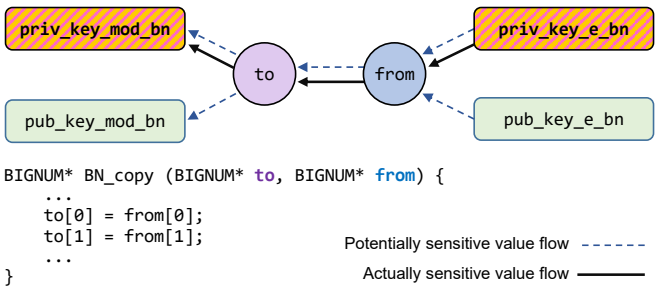

Fig. 4: Example of potentially sensitive (dashed arrows) and actually sensitive (solid arrows) value flows. Sensitivity labels are maintained using dynamic DFT to distinguish between the two. DynPTA uses these labels to decide whether the object written through the `to` pointer must first be encrypted or not.

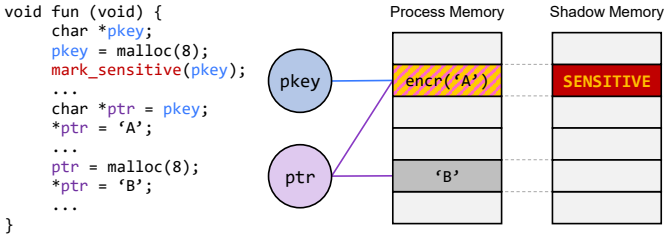

Fig. 5: In this example, `ptr` is a partially sensitive pointer that can point to both sensitive and non-sensitive data. By keeping sensitivity labels in a shadow memory, DynPTA can selectively apply the required cryptographic transformations only when the pointer dereference involves sensitive data.

deciding to apply the predicated transformation described in Section IV-D2, once the sensitive label is applied to a memory location, all memory instructions operating on that memory location are automatically instrumented with AES operations.

Because our system relies on runtime DFT label information, if the same load–store chain (which includes indirect memory accesses) is invoked multiple times with sensitive and non-sensitive values, the sensitive labels will be propagated *only* to the intended targets of the sensitive value flows. Figure 4 illustrates this case using a simplified code snippet from the OpenSSL library. The function `BN_copy` is invoked for the processing of both the private and the public SSL key. The `from` and `to` pointer arguments can point to parameters of both the public and the private key, but only the latter needs to be protected. Based on the label of a given object, DynPTA decides whether the object must be encrypted or not before writing it in memory through the `to` pointer.

Sensitive labels are retained for the lifetime of the object. Sensitive heap objects have their labels cleared when the object is freed via the `free` Libc function. Similarly, sensitive labels associated with local variables (allocated on the stack) are cleared when the function returns.

2) Runtime Handling of Potentially Sensitive Pointers: To overcome the overapproximation of points-to analysis and avoid costly cryptographic operations for non-sensitive data, we instrument the dereferences of partially sensitive pointers to perform a shadow memory lookup, and decide at runtime whether to apply the cryptographic transformation or not, as shown in Figure 5. Absence of a label indicates that the accessed memory location is not sensitive, in which case the expensive cryptographic operations are elided, and the original memory load or store operation is performed directly. In case of loops operating incrementally over potentially sensitive pointers, we further optimize their label lookups as discussed in Appendix D.

We should stress that Steensgaard's analysis identifies only a *fraction* of all memory accesses as potentially sensitive, and only these are instrumented with label lookups. At runtime, only the fraction of potentially sensitive memory operations that *truly* access sensitive objects undergo the expensive AES transformations. Indicatively, our evaluation shows that about 15% of all memory operations in the tested programs are instrumented with label lookups, and at runtime, only 1–5% of all memory accesses undergo AES transformation.

E. In-memory Data Protection using Encryption

Sensitive data remains encrypted in memory as long as it flows within DynPTA's *protection domain*. This domain depends on the code that takes part in DynPTA's whole-program analysis, on which points-to analysis is performed. If sensitive data has to flow to an external library that is not part of the protection domain, then for compatibility reasons DynPTA first decrypts the data. At that point, the plaintext form of sensitive data will exist in memory, and could be leaked due to some vulnerability. This is the main reason we require whole program analysis (including external libraries), to ensure that DynPTA's protection domain spans the whole (to the extent possible) code base of the application. Based on our experiments with various applications and use cases (Section VI), we did not encounter and could not identify any other situation in which sensitive data should escape the protection domain.

Similarly to our previous work [33], we use AES-128 in Electronic Code Book (ECB) mode to ensure the confidentiality of sensitive data in memory. Modern processors offer hardware-accelerated AES operations, such as the AES-NI extensions of Intel processors, on which we rely to improve performance.

AES-128 has 10 rounds of operations for both encryption and decryption. Each of these rounds has its own "round keys" that are generated from the initial secret key. To avoid the overhead of generating the round keys from scratch before each AES operation, DynPTA pregenerates them from the initial secret key and stores them in registers. Modern Intel and AMD processors support SSE [46] and provide 16 128-bit registers (XXM0–XMM15). We use these registers to store the *expanded* round keys for all ten encryption round keys. Decryption round keys are the inverse of the encryption round keys, and Intel provides the `aesimc` instruction to efficiently compute them. Applications that rely on XMM registers for computation are not directly compatible with DynPTA. This is not a major issue, however, because most such applications have the option of being compiled without SSE support for backwards compatibility reasons.

V. Implementation

We implemented DynPTA on top of LLVM 7.0 [47]. As DynPTA needs to perform whole-program analysis on the application and its dependent libraries, we use link time optimization (LTO) with the Gold linker [48]. We include all imported libraries in our analysis except Glibc, for which instead we provide our own implementation of commonly used functions (e.g., `memcpy`, `memcmp`, `strcpy`). Our observation is that sensitive data is not passed to other Libc functions, but additional ones can be supported as needed. We modified the build scripts of the applications and libraries to use the LLVM tools (`clang`, `llvm-ar`, and `llvm-ranlib`), which operate on LLVM's intermediate representation (IR), instead of their counterparts from the GCC toolchain.

A. Context-sensitive Heap Modeling

The first step for modeling a context-sensitive heap is to identify all memory allocation wrappers (as discussed in Section IV-B), for which we have implemented an LLVM pass. For functions that return pointers, we use the *intraprocedural* Andersen's points-to analysis provided by LLVM (`CFLAAAnders`), to determine if the function returns a pointer to memory allocated from within the function. Being intraprocedural, this is a lightweight and inexpensive analysis with few constraints, and we can thus afford to use the more expensive (but more precise) Andersen's algorithm.

As discussed in Section IV-B1, we must iteratively analyze functions to identify nested wrappers. We set the iteration limit for this process to five, which is more than enough for the tested applications. Once the wrappers are identified, we generate their summary and insert it at the respective call sites.

B. Steensgaard's Analysis

We implemented our Steensgaard's pointer analysis on top of SVF [36], a popular static analysis framework, as an LTO pass. SVF supports multiple variants of Andersen's algorithm, but does not support Steensgaard's algorithm.

SVF operates on the LLVM IR representation by iterating over every IR instruction and capturing their pointer constraints. We solve each of the constraints collected, performing set unification operations when required, as described in Section II-A1. That is, when we discover a new points-to target t for a pointer p, we unify the sets T and P, where $t \in T$, and P is the set of objects that p points to. The details are provided in Appendix C. Solving these constraints results in computing all points-to relationships associated with the constraints. Only when solving a constraint results in the derivation of a new call target for an indirect function call, the constraints associated with the newly discovered target must be recomputed. Apart from this, every constraint is processed exactly once, allowing the algorithm to operate in almost linear time.

We use SVF's interfaces to export the analysis results. This allows our implementation to be seamlessly used as a replacement for the other variants of Andersen's analysis provided by SVF (we are in the process of contributing our Steensgaard's analysis implementation to the SVF project).

C. Static Value Flow Analysis

As discussed in Section IV-C, objects marked as sensitive may be copied and stored to other objects and variables. The LLVM instructions `LoadInst` and `StoreInst` are used to read from and write to memory, respectively. To identify sensitive value flows, we track the flows that begin from a `LoadInst` reading a sensitive object, and terminate in a `StoreInst` writing to a non-sensitive object.

As discussed earlier, indirect value flows via pointers are possible, and we use the Steensgaard's analysis results to resolve the sources and targets of any pointers involved in indirect value flows. Due to its inherent overapproximation, this means that the sink sites of some of the identified value flows may *not* receive any sensitive values at runtime—this is the reason for introducing dynamic data flow tracking to maintain sensitivity labels. To aid the DFT phase identify these *potentially sensitive* value flows, we add metadata to every instruction that is part of them.

D. Scoped Dynamic Data Flow Tracking

Similarly to existing DFT frameworks [49], DynPTA maintains a shadow memory located at a fixed offset in the process' address space, which keeps a sensitivity label for each byte of process data. To speed up label initialization and lookup, we use hand-crafted assembly code. Note that the shadow memory does not have to be kept secret from the attacker.

The set of tracked memory objects (located on the stack, heap, or the global section) includes the objects annotated directly by the programmer, as well as the rest of the objects derived through value flow analysis. At program startup, the only memory locations labeled as sensitive are the locations that are explicitly marked by the programmer using DynPTA's `mark_sensitive()` function. Marking a memory location as sensitive i) applies the sensitive label to it, and ii) encrypts the existing data at that location. From that point on, our scoped DFT logic propagates sensitive labels only for instructions that contain our inserted metadata (Section V-C), i.e., the instructions that take part in potentially sensitive value flows.

If a `LoadInst` reads from a memory location marked as sensitive, the location of the terminating `StoreInst` is also labeled as sensitive. At that point, we insert an LLVM `BranchInst` that checks if the value about to be stored is marked as sensitive, in which case it encrypts the value before storing it. Any further operations on this object will always undergo AES transformation, as the label is maintained for the lifetime of the object. In this way, we apply AES *only* to objects that are sinks for truly sensitive value flows.

In addition to properly maintaining sensitive value flows, Steensgaard's analysis provides us with every `LoadInst` and `StoreInst` that may access sensitive data. For each of these instructions, we again perform a label lookup to determine whether the memory operand is actually sensitive. In that case, an LLVM `BranchInst` invokes the corresponding AES operations—otherwise the memory access proceeds normally.

TABLE I: Applications used for performance evaluation.

Application	Protected Data	KLOC	Bitcode Size	DynPTA Compilation
Nginx + OpenSSL	Private Key	389	8M	50.6 min
Httpd	Password	179	3.7M	11.0 min
Lighttpd + ModAuth	Password	83	1.9M	2.8 min
MbedTLS server	Private Key	54	726K	1.3 min
OpenVPN	Private Key	329	3.5M	59.1 min
Memcached + Auth.	Password	71	1.1M	1.0 min
ssh-agent	Private Key	52	640K	1.3 min
Minisign	Private Key	45	1.2M	37 sec

VI. Performance Evaluation

We evaluated DynPTA with a set of eight popular applications. In each case, we annotate sensitive data such as passwords and private keys to be protected. Every experiment is performed 20 times, and we report averages. We run all our applications under test on a machine with an Intel Core i7-6700 CPU and 32 GB of RAM, running Ubuntu 19.10 and Linux kernel 5.3.0-40. For server–client experiments, we run the client on a machine with an Intel Xeon E5-2620 CPU and 64 GB of RAM, running Ubuntu 18.04 and Linux kernel 4.15.0-106. Both the server and client machines were on the same local 1Gbit/s network.

A. Applications

Table I lists the applications used in our evaluation, and the respective data annotated as sensitive. We included popular web servers, VPN servers, and desktop utilities. We also report the number of source code lines, the LLVM bitcode size, and the time that DynPTA takes to generate the hardened binaries.

Nginx: We built Nginx with the `HTTP_SSL_module` enabled and linked it with the OpenSSL library. We use LLVM's link time optimization (LTO) to generate the combined bitcode that includes the main Nginx executable and all libraries. Our use case for DynPTA is to protect the parameters of the SSL private key. These are in `BIGNUM` objects, which are referred to by pointers stored in the `rsa` field of the `pkey` object. The function `ssl_set_pkey` initializes these pointers, which we mark as sensitive. As shown in Table I, the use of Steensgaard's algorithm [37] allows DynPTA to complete in less than an hour all its analysis and instrumentation passes. Indicatively, an Andersen's pass *alone* for the same code requires almost 11 hours to complete.

Apache Httpd with Authentication: We used LTO to link Httpd statically with Apache's Portable Runtime (APR). Httpd supports password protection for certain directories through the ModAuth module. The password is stored on the heap and is referred to through the pointer `file_password` (in `mod_authn_file.c`), which we annotate as sensitive. This object is allocated via the wrapper `ap_getword()`, provided by APR. Our context-sensitive heap modeling successfully identifies this function as a memory allocation wrapper.

Lighttpd: Similarly to Httpd, Lighttpd also supports ModAuth for password-protecting files and directories. The pointer `password_buf` in function `mod_authn_file_htpasswd_get` is initialized to store the address of the password, and we annotate it as sensitive.

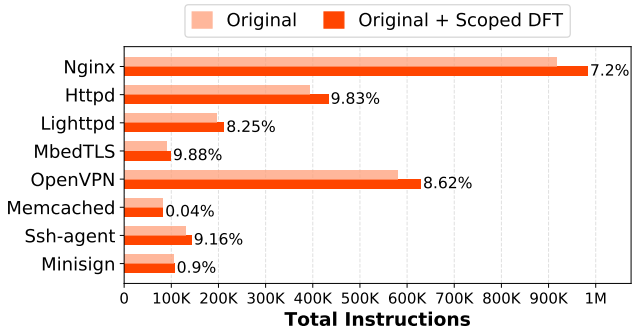

Fig. 6: New instructions added due to scoped DFT for potentially sensitive value flows.

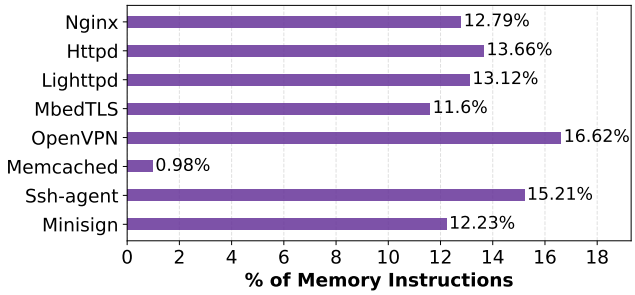

Fig. 7: Percentage of protected memory instructions.

MbedTLS server: MbedTLS is a lightweight TLS library which also provides a simple TLS server. Similarly to OpenSSL, MbedTLS uses a custom data type to represent multi-precision integers called `mbedtls_mpi`. The SSL private key is stored within `mbedtls_rsa_context`, in objects of type `mbedtls_mpi`, which we annotate as sensitive.

OpenVPN: We configured OpenVPN to work with OpenSSL certificates, and used LTO to build the combined LLVM IR bitcode. Similarly to Nginx, we annotate the parameters of the SSL private keys as sensitive.

Memcached with Authentication: When Memcached is compiled with LibSASL, client connections can be protected with a password. The variable `buffer` in function `sasl_server_userdb_checkpass` stores this password, which we mark as sensitive.

ssh-agent: Private SSH keys are typically password-protected on disk, and `ssh-agent` conveniently keeps them in memory so that users do not have to re-type the password. The private key is stored in an object of type `ssh_key` (initialized in function `sshkey_new`), which we mark as sensitive.

Minisign: Minisign is a simple file signing tool that uses Libsodium for hashing and signing. The private key used for signing is stored in an object of type `SeckeyStruct`, which we mark as sensitive.

B. Scoped Data Flow Tracking

1) Static Instrumentation: As discussed in Section IV-D, DynPTA uses *scoped* DFT to track sensitive value flows and maintain labels for sensitive data. Figure 6 shows the

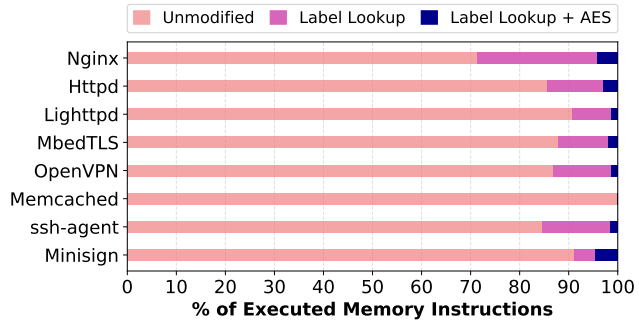

Fig. 8: At runtime, the vast majority of memory accesses proceed normally. Label lookups are performed only for up to 24% of all memory accesses (Nginx), and only a fraction of those (up to 4% for Nginx) involve AES operations.

percentage of new instructions added for scoped DFT, compared to the original program. Among the evaluated applications, the maximum percentage of additional instructions for the DFT logic is only 9.08%.

Similarly, only a *fraction* of all memory load and store instructions have to be instrumented to protect sensitive data. The instrumentation in this case consists of a lightweight shadow memory lookup, which invokes the AES transformation in case the data is indeed sensitive. Figure 7 shows the percentage of memory instructions that are instrumented for data protection. In the worst case (OpenVPN), only 16.62% of all memory operations have to be instrumented.

2) Runtime Performance Benefit: Without scoped DFT, all protected memory accesses (Figure 7) would *always* have to undergo expensive AES transformation. By introducing a lightweight label lookup, AES can be avoided when the accessed data turns out to be non-sensitive.

To assess the performance benefit of this approach, we first compare the cost of a shadow memory lookup with the cost of the AES data transformation using a microbenchmark. We performed three experiments by instrumenting one billion single-byte memory accesses with i) a label lookup, ii) AES encryption, and iii) AES decryption, which took 3.3, 14.2, and 16.5 seconds to complete, respectively. This means that the cost of AES encryption and decryption is *at least* 430% and 500% that of a label lookup.

Then, we use a custom Pin [50] tool to record how many memory accesses involve label lookups, and among those, how many perform AES operations on the accessed data. As shown in Figure 8, shadow memory lookups are performed only for up to 24% of all memory accesses, while only up to 4% of them undergo expensive AES cryptographic operations.

Without scoped DFT, *all* protected memory accesses would always involve AES, resulting in a prohibitively high runtime overhead. To demonstrate this, we applied DynPTA *without* scoped DFT on MbedTLS. This required significant amount of effort because applying AES to *all* potentially sensitive memory instructions identified by the pointer analysis and value flow analysis involves many unintended objects, such as file handles and network sockets. These objects are passed directly to Libc interfaces and used by the kernel. Ensuring that every Libc or kernel interface appropriately decrypts (and re-encrypts) these objects requires significant engineering effort and therefore we did not attempt it for the rest of the (more complex) applications. When removing scoped DFT, the runtime overhead for running MbedTLS server increases from 4.1% to 56%. We provide the details of this experiment and the rest of our performance evaluation results in the following section.

C. Runtime Overhead

1) Real-world Use Cases: To evaluate the runtime overhead of DynPTA, we harden the applications listed in Table I to protect their sensitive data (listed in the second column), and drive them using various workloads. For all applications we use their default configuration. For web servers (Nginx, Httpd, Lighttpd) we use ApacheBench [51] to perform five rounds of 10,000 requests, with each round requesting a file of increasing size (from 4KB to 1MB).

Figures 9(a)–9(c) show the overhead of DynPTA when protecting the TLS key for Nginx and the authentication password for Httpd and Lighttpd. In all cases, the in-memory protected objects are accessed only during connection establishment, and the AES transformations are performed only at that time. We observe the highest overhead (19%) for Nginx, because the TLS handshake involves multiple complex operations to derive a new session key from the (protected) TLS private key per connection. In contrast, password-based authentication involves a one-time decryption and a short sequence of byte-by-byte comparisons of the user-provided password with the password on file. This results in a lower overhead ranging from 6.5% in the worst case for the shortest response size, to an amortized 1.86% for 1MB responses. Nginx's overhead is not amortized as the response size increases, because many label lookups (proportional to the response size) still have to be performed (as shown in Figure 8, Nginx has at least twice as many label lookups compared to other applications).

For the rest of the applications (Figure 9(d)), we used a variety of workloads. For the MbedTLS server, we used its TLS client to perform 100,000 requests for a 4KB file over the same connection (default behavior), which has also been the main use case in previous selective data protection works [31, 33]. The overhead in this case is just 4%, as the protected private key is used only during the initial connection establishment. Indicatively, although this result is not directly comparable to previous works due to the different experimental environment, for the same server application, workload, and protected data, the reported overhead for in-memory encryption [33] is 13% and for DataShield [31] is 35%.

For OpenVPN, we downloaded a 100KB file 10,000 times over a VPN connection using ApacheBench, observing an overhead of 10.47%. Similarly to Nginx, although most of the expensive AES operations happen during connection establishment, there is still a significant amount of label lookups throughout the whole duration of the experiment. For Memcached, we used its benchmarking tool Mutilate [52] to

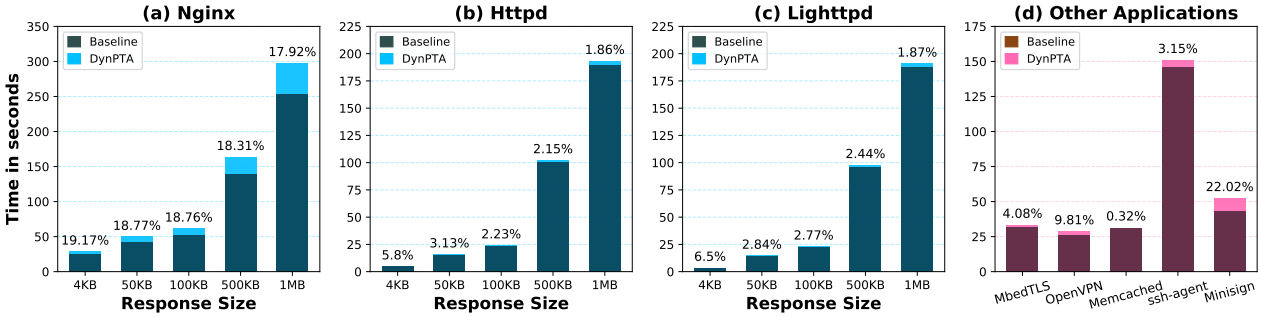

Fig. 9: Runtime overhead of DynPTA for popular web servers (a)–(c) and other applications (d).

generate five billion operations with its default configuration (get/set ratio of 0.5, key size of 30 bytes, value size of 400 bytes). DynPTA's overhead in this case is negligible (0.32%), because the variable that stores the protected password is not pointed to by any pointer and is not copied to any other variable, requiring only a fraction (about 1%) of memory accesses to be protected, as also shown in Figure 8.

For the two client-side utilities, we performed 500 logins to another host in the same subnet that triggered ssh-agent, and signed a 1GB file using Minisign. The overhead for ssh-agent is just 3.15%, while Minisign exhibits the highest overhead among all our use cases at 22%. Minisign operates by first pre-hashing the file and then signing the hash value byte-by-byte, with every iteration of the signing loop requiring a decryption of the private key, resulting in such a high overhead—which though is expected as a fully compute-bound use case.

2) Increasing the Amount of Sensitive Data: The key insight behind selective data protection, and DynPTA in particular, is that instead of protecting all data by spending as few extra CPU cycles per memory access as possible, we protect only data that is really security-critical, and thus afford to spend more CPU cycles for only a fraction of memory accesses. As expected, however, any performance benefits will diminish as the amount of protected data increases, and for this reason we performed some additional experiments to explore this tradeoff.

We used MbedTLS to explore a worst-case scenario by marking additional *non-critical* data as sensitive. Specifically, besides the SSL private key, we progressively mark other fields of the `mbedtls_ssl_context` data structure as sensitive. These include SSL handshake parameters, configuration options, and input/output buffers. In each round we mark more fields as sensitive, until the whole data structure is marked as sensitive.

Figure 10 shows how the overhead increases modestly from 4% to 11% in the first four measurement rounds, as we keep marking mostly configuration-related fields as sensitive. Marking the input and output buffers as sensitive in the final two rounds increases the overhead considerably to 46%, because these buffers are used as part of every transmission, in contrast to the private key and the rest of the fields, which are accessed only during the TLS handshake.

Besides MbedTLS, we also experimented with Nginx by enabling HTTP password authentication and protecting the in-memory passwords (in addition to the SSL private key),

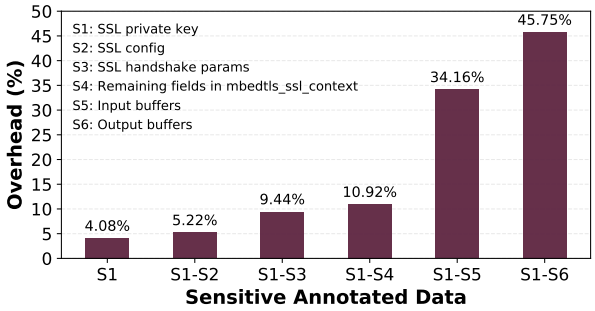

Fig. 10: Runtime overhead of MbedTLS for an increasing amount of protected (non-critical) data.

observing only a minor increase of 1% in the overall performance overhead. We discuss in detail this experiment in Appendix F. Finally, we also performed some microbenchmarks to further study the benefits of scoped DFT as the percentage of sensitive data in the program increases, the results of which we provide in Appendix G. Our main finding is that once sensitive data exceeds 70–80% of all data, the scoped DFT and label lookups become more costly than simply encrypting all objects identified by the points-to analysis.

VII. Security Evaluation

A. Heartbleed

Heartbleed [5] is a heap overflow vulnerability due to a missing bounds check in the TLS Heartbeat feature of OpenSSL. An attacker can send a malicious request that causes a buffer over-read in the server's memory and allows the leakage of sensitive data, including the private server SSL keys, back to the attacker through the generated response.

We compiled Nginx with OpenSSL v1.0.1f and verified that the PoC exploit [53] was indeed capable of leaking the private TLS key. We observed that the leakage of the key was dependent on the heap allocations, that is, the address of the private key and the address of the vulnerable request buffer that is over-read. The private key is initialized during server startup and typically occupies a low address on the heap. To leak the private key, the vulnerable request buffer must be allocated *below* this address. During experimentation, we observed that there are "holes" below the address of the private key on the

heap that occasionally would be allocated to the vulnerable request buffer, allowing the exfiltration of the private key.

We then marked the private key as sensitive, as described in Section VI-A, and hardened the server using DynPTA. Using the above PoC exploit, we repeatedly verified that whenever the private key was leaked, it was always encrypted.

B. Spectre

Transient execution vulnerabilities allow the leakage of otherwise inaccessible data from memory, and are thus another class of attacks DynPTA can defend against. We evaluated DynPTA against this type of attacks using two Spectre [18] variants for which we could obtain PoC exploits [40, 41].

Intel CPUs contain a *pattern history table (PHT)* that uses the history of past taken/not-taken branches for branch prediction. The Spectre-PHT variant poisons the PHT, causing mispredictions in the direction of conditional branches, which can be used by attackers to bypass bounds checks in the program, and speculatively load sensitive data into the cache. From there, data can be leaked via various cache side-channel attacks. The Spectre-PHT PoC [40] contains a bounds check which is bypassed to leak a secret string.

Besides the PHT, CPUs also contain a *branch target buffer (BTB)* that uses the history of past branch targets for branch target prediction. The Spectre-BTB variant poisons the BTB to steer transient execution to special "gadgets" found in the program, which can be used to leak sensitive data. Similarly to the previous exploit, the Spectre-BTB PoC [41] contains a secret string that is leaked by redirecting the speculative execution to an appropriate gadget.

For both PoCs, we marked the secret string as sensitive, and used DynPTA to harden the exploit program (more details and the code for both PoCs are provided in Appendix E). When the string is speculatively accessed, its *encrypted* form is loaded in the cache. Therefore, the confidentiality of the string is always preserved when being leaked through a cache side channel.

VIII. Limitations and Discussion

a) Performance Optimizations: Although DynPTA allows us to scale selective data protection to larger applications with modest overhead, there is still opportunity for further optimizations that will lower the overhead even further. Label lookups can disrupt cache locality, resulting in a higher number of cache misses. We plan to investigate this issue further and adapt the shadow memory implementation accordingly.

DynPTA performs context-sensitive modeling only for heap analysis. Other regions in the program code, such as code hotspots and critical objects, could also benefit from selective, summary-based, context sensitivity. Smaragdakis et al. [54, 55] discussed selective context sensitivity with respect to Java programs. Similarly, Sridharan et al. [56] proposed refinement-based context sensitive pointer analysis. In their current form, these techniques are applicable only to Java programs, but we plan to investigate their adaptation for C/C++ programs.

Our DFT-based optimization is not limited to Steensgaard's algorithm, and can improve the precision of *any* static pointer analysis algorithm. In particular, TeaDSA [57] is a promising unification-based pointer analysis algorithm that aims to limit oversharing and thus improve scalability. Despite our efforts, however, we could not successfully use it to run larger applications such as Nginx with OpenSSL.

Iodine [58] successfully uses profiling to improve the performance of DFT. Similarly, various works have presented techniques to optimize dynamic flow tracking [59, 60, 61, 62, 63]. We plan to investigate the application of these techniques to improve the performance of our system.

b) Ensuring Data Integrity: DynPTA protects all memory operations to sensitive objects with strong AES encryption. Encryption is not enough though to fully guarantee data integrity, as the attacker may be able to swap encrypted objects, or corrupt existing values (altering protected data with arbitrary values is still not possible, as the encryption key remains inaccessible to the attacker) [33]. To that end, we plan to extend our data protection mechanism with an HMAC-based scheme to ensure data integrity.

c) Leaking Register Contents via Vector Register Sampling: Vector Register Sampling [64] is a recent speculative execution vulnerability that might allow partial data values to be leaked from vector registers under certain microarchitectural conditions. Although this vulnerability could affect the security of our system, as we rely on vector registers to store the AES round keys, it was patched via a microcode update [65].

IX. Related Work

Data-only attacks were introduced more than a decade ago [1], but have only recently started gaining popularity [2, 3, 4, 6, 66, 67]. On the other hand, transient execution attacks such as Spectre [18] are more recent, and can leak secrets from a process's memory through microarchitectural side channels. In the following, we discuss various types of defenses that can be used against these attacks.

a) Memory Safety: Defenses based on memory safety ensure that all pointers access their intended referents, thus ensuring spatial safety. SoftBound [14] and CCured [68] maintain bounds information for each pointer and ensure spatial safety by performing bounds checks during all pointer dereferences. AddressSanitizer (ASan) [69] and Baggy Bounds Checking [15] associate metadata with each object and detect out-of-bounds memory accesses. In general, defenses based on memory safety use whole-program instrumentation to protect *all* program data and require *every* memory instruction to be instrumented with bounds checks. Therefore, they incur a very high runtime overhead. Moreover, these techniques do not protect against transient execution attacks.

DataShield [31] enforces memory safety at an object granularity by partitioning process memory into sensitive and non-sensitive regions. It then performs fine-grained bounds checks for sensitive pointers and coarse-grained bounds checks for non-sensitive pointers. Similarly, ConfLLVM [70] partitions the memory into private and public regions and ensures that every pointer points to its own memory region. Moreover, ConfLLVM requires the programmer to classify all arguments

of a function as public or private, whereas we only require annotating the initial sensitive data. As these are software-based defenses, they can protect against data leakage attacks, but not against transient execution attacks.

b) Data Flow Integrity: Dataflow Integrity (DFI) [16] ensures that all memory accesses adhere to valid data flow paths identified by static analysis. Therefore, any static analysis imprecision results in false negatives. DFI requires *every* memory instruction to be instrumented with software checks, leading to high overhead (up to 104% for SPEC [71]). Hardware-based DFI techniques [72, 73] have lower overhead, but require custom hardware. Moreover, DFI cannot protect against transient execution attacks.

c) Isolation-based Defenses: Many works rely on memory isolation to protect security-critical data [30, 32, 42, 74, 75, 76, 77, 78]. Glamdring [32] moves all sensitive-annotated data into SGX enclaves, and uses static dataflow analysis [79] and static backward slicing [80] to transform all functions that may access the sensitive data to use the appropriate SGX entry and exit routines. ERIM [42] and LibMPK [81] provide hardware-enforced isolation for sensitive code and data using Intel Memory Protection Keys (MPK) [82]. However, Intel MPK is vulnerable to transient execution attacks [7], therefore these solutions cannot protect against them. Donky [83] is a hardware-software codesign for the RISC-V [84] Ariane CPU, offering strong in-process isolation based on memory protection domains. The xMP [30] system relies on Xen [85] to protect selective sensitive data. However, the programmer has the burden of manually inserting the xMP domain switches, making the process cumbersome and error-prone. Overall, unlike DynPTA, which provides fine-grained protection, these isolation-based approaches provide page-level protection and this requires refactoring the data layout.

Ginseng [74] ensures that sensitive data is always stored in registers and relies on ARM TrustZone to protect against an untrusted operating system. PT-rand [75] protects kernel page tables by randomizing and hiding their locations.

For Android applications, FlexDroid [86] introduces an isolation mechanism that provides fine-grained access control for third-party Android libraries. On the browser front, privilege separation techniques such as Chrome's Site Isolation [87] and Firefox's RLBox [88] are being widely deployed [89].

Various works have presented techniques to assist program partitioning and privilege separation [21, 90, 91, 92, 93, 94]. These techniques cannot be easily applied to the problem of tracking sensitive memory operations because they are specific to privilege separation. PtrSplit [92] is a type-based technique that allows the use of only intra-procedural analysis instead of requiring global interprocedural analysis. However, it assumes that `void` pointers are not used as function arguments. Based on experience with codebases such as OpenSSL, this assumption does not always hold.

d) In-memory Transformation: Defenses based on in-memory transformation [33, 95] change the representation of memory-resident objects using encryption. Data space randomization [95] transforms in-memory data using simple XOR, and was originally designed to defend against code injection attacks. It thus cannot prevent data leakage, as the memory-resident XOR keys can be leaked as well, and the XOR transformation can be reversed. CoDaRR [96] extends DSR to periodically rerandomize the masks used to provide probabilistic guarantees against disclosure attacks, but suffers from the same weaknesses as DSR due to the use of XOR. HARD [97] is an ISA extension to the RISC-V architecture to support DSR at the hardware level.

e) Defenses against Transient Execution Attacks: Retpoline [98] mitigates Spectre [18] by hardening all branch instructions against speculative execution. ConTExT [99] proposes a backwards-compatible architectural change that mitigates transient execution attacks. SpecFuzz [100] performs fuzzing to determine which branches are benign and which can lead to speculative execution, and removes hardening from the benign ones, thus lowering the overhead. SPECCFI [101] proposes a hardware extension that uses CFI [102] to determine whether speculative execution targets a legal destination or not. Blade [103] stops the leakage of sensitive data via speculative execution by cutting the dataflow (e.g., using memory fences) from expressions that speculatively introduce secrets, to those that leak them through the caches. Swivel [104] hardens WebAssembly [105] applications against Spectre attacks.

The above defenses focus on hardening all or a subset of all branches in a program, without considering whether speculative execution might actually leak sensitive data. In comparison, DynPTA focuses on preventing the leakage of only sensitive data, obviating the need for the above mitigations.

X. Conclusion

DynPTA combines static and dynamic analysis to provide a practical defense against data leakage attacks due to memory disclosure or transient execution vulnerabilities. DynPTA requires developers to just mark certain objects in the program's memory as sensitive, and automatically derives all sensitive memory operations, which are then protected using encryption. To ameliorate the inherent overapproximation of static pointer analysis, DynPTA uses a scoped form of data flow tracking that maintains sensitivity labels and tracks their flow over the set of instructions identified by the pointer analysis. This allows DynPTA to ensure the confidentiality of sensitive data in real-world applications with modest overhead. As part of our future work, we plan to implement support for data integrity, and also investigate further optimizations to our scoped DFT logic that will reduce the runtime overhead even further.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback, and our PC point of contact, Yajin Zhou, for helping us revise our manuscript. We also thank Hamed Ghavamnia for his valuable comments on an earlier draft of this paper. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045.

REFERENCES

[1] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[2] D. Moghimi, "Subverting without EIP," https://moghimi.org/blog/subverting-without-eip.html, 2014.

[3] F. Falcon, "Exploiting adobe flash player in the era of control flow guard," in *Black Hat Europe*, 2015.

[4] B. Sun, C. Xu, and S. Zhu, "The power of data-oriented attacks: Bypassing memory mitigation using data-only exploitation," in *Black Hat Asia*, 2017.

[5] "The heartbleed bug," https://heartbleed.com/, 2020.

[6] R. Rogowski, M. Morton, F. Li, K. Z. Snow, F. Monrose, and M. Polychronakis, "Revisiting browser security in the modern era: New data-only attacks and defenses," in *Proceedings of the 2nd IEEE European Symposium on Security & Privacy (Euro S&P)*, April 2017.

[7] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 249–266.

[8] J. Voisin, "Spectre exploits in the "wild"," https://dustri.org/b/spectre-exploits-in-the-wild.html, 2020.

[9] "Virus total: Spectre exploit"," https://www.virustotal.com/gui/file/6461d0988c835e91eb534757a9fa3ab35afe010bec7d5406d4dfb30ea767a62c/detection, 2021.

[10] S. Rottger and A. Janc, "Leaky page: Spectre proof-of-concept for Chrome browser," https://leaky.page/, 2021.

[11] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2008, pp. 263–277.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2002, pp. 275–288.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler-enforced temporal safety for C," in *Proceedings of the International Symposium on Memory Management (ISMM)*, 2010, pp. 31–40.

[14] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.

[15] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, 2009, pp. 51–66.

[16] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 147–160.

[17] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Everything you want to know about pointer-based checking," in *Proceedings of the 1st Summit on Advances in Programming Languages, (SNAPL)*, 2015, pp. 190–208.

[18] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, May 2019.

[19] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," in *Proceedings of the 6th IEEE European Symposium on Security & Privacy (EuroS&P)*, 2021.

[20] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, 2003.

[21] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 689–703.

[23] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library OSes for multi-process applications," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, 1993, pp. 203–216.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[26] 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 (CCS)*, 2011.

[27] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017, pp. 437–452.

[28] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 1607–1619.

[29] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *ACM International Conference on Virtual Execution Environments (VEE)*, 2008, pp. 71–80.

[30] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 584–598.

[31] S. A. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 193–204.

[32] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017, pp. 285–298.

[33] T. Palit, F. Monrose, and M. Polychronakis, "Mitigating data leakage by protecting memory-resident sensitive data," in *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, 2019, pp. 598–611.

[34] Intel, "Intel software guard extensions," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, 2020.

[35] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[36] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[37] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32–41.

[38] M. Hind and A. Pioli, "Which pointer analysis should I use?" in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000, pp. 113–123.

[39] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[40] "Proof of concept – Spectre variant 1," https://github.com/crozone/SpectrePoC, 2020.

[41] "Proof of concept – Spectre variant 2," https://github.com/Anton-Cao/spectrev2-poc, 2020.

[42] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1221–1238.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[44] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 264–274.

[45] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," *ACM Sigplan Notices*, vol. 30, no. 6, pp. 1–12, 1995.

[46] "Streaming SIMD extensions," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2020.

[47] "The LLVM compiler infrastructure," https://llvm.org/, 2020.

[48] "GNU binutils," https://sourceware.org/binutils/, 2020.

[49] "DataFlowSanitizer," https://clang.llvm.org/docs/DataFlowSanitizer.html, 2020.

[50] "Pin 3.2 user guide," https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/, 2020.

[51] "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2020.

[52] "Mutilate – a memcached load generator," https://github.com/leverich/mutilate, 2020.

[53] "Heartbleed proof-of-concept," https://github.com/mpgn/heartbleed-PoC, 2020.

[54] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 2, pp. 1–40, 2020.

[55] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 423–434, 2013.

[56] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 387–400, 2006.

[57] J. Kuderski, N. Lê, A. Gurfinkel, and J. Navas, "TeaDsa: Type-aware DSA-style pointer analysis for low level code," *FMCAD*, 2018.

[58] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, "Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 490–504.

[59] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant, "DECAF: A platform-neutral whole-system dynamic binary analysis platform," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, 2016.

[60] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 31–45.

[61] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[62] J. Galea and D. Kroening, "The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 622–636.

[63] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[64] Intel, "Vector register sampling / cve-2020-0548 , cve 2020-8696 / intel-sa-00329," https://software.intel.com/security-software-guidance/advisory-guidance/vector-register-sampling, 2021.

[65] Intel, "Microcode update guidance," https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/sa00329-microcode-update-guidance.pdf, 2021.

[66] 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 *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 969–986.

[67] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 1868–1882.

[68] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.

[69] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 309–318.

[70] A. Brahmakshatriya, P. Kedia, D. P. McKee, D. Garg, A. Lal, A. Rastogi, H. Nemati, A. Panda, and P. Bhatu, "ConfLLVM: A compiler for enforcing data confidentiality in low-level code," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[71] "SPEC CPU," https://www.spec.org/benchmarks.html#cpu, 2020.

[72] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 1–17.

[73] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[74] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[75] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[76] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library os for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658.

[77] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves." in *NDSS*, 2017.

[78] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 264–278.

[79] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[80] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*. ACM, 1981, pp. 439–449.

[81] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for Intel memory protection keys (Intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[82] J. Corbet, "Memory protection keys," https://lwn.net/Articles/643797/, 2015.

[83] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys–efficient in-process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.

[84] "RISC-V: The free and open RISC instruction set architecture," https://riscv.org/, 2020.

[85] Xen, "The xen hypervisor," https://xenproject.org/developers/teams/xen-hypervisor/, 2020.

[86] I. Shin and J. Seo, "FlexDroid: Enforcing in-app privilege separation in Android," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[87] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1661–1678.

[88] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *29th USENIX Security Symposium (USENIX) Security 20)*, 2020, pp. 699–716.

[89] Nathan Froyd, "Securing Firefox with WebAssembly," https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2021.

[90] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 323–333.

[91] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *27th USENIX Security Symposium, (USENIX Security 18)*, 2018, pp. 65–82.

[92] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2359–2371.

[93] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1023–1040.

[94] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The art of app compartmentalization: Compiler-based library privilege separation on stock Android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1037–1049.

[95] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008, pp. 1–22.
[96] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "CoDaRR: Continuous data space randomization against data-only attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 494–505.
[97] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek *et al.*, "Hardware assisted randomization of data," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 337–358.
[98] "Retpoline: A branch target injection mitigation," https://software.intel.com/security-software-guidance/deep-dives/deep-dive-retpoline-branch-target-injection-mitigation, 2021.
[99] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A generic approach for mitigating Spectre," in *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20). Internet Society, Reston, VA*, 2020.
[100] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface," in *Proceedings of the 29th USENIX Security Symposium*, 2020.
[101] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SPECCFI: Mitigating Spectre attacks using CFI informed speculation," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.
[102] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2005, pp. 340–353.
[103] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–30, 2021.
[104] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *Proceedings of USENIX Security Symposium*, 2021.
[105] "WebAssembly (WASM)," https://webassembly.org/, 2020.

APPENDIX

A. Imprecision Introduced by Steensgaard's Algorithm

To compare the precision of Steensgaard's pointer analysis algorithm with that of Andersen's algorithm, we consider the function `ngx_rbtree_rotate` from Nginx's codebase. This function accepts as input an argument `root`, which is of type `ngx_rbtree_node**`. Two different call sites invoke this function with two different arguments (`cache->rbtree` and `cf->cycle->conf->rbtree`). Figure 11 shows how the use of Steensgaard's analysis in this case leads to imprecision. Because Steensgaard's algorithm uses unification to resolve constraints, it *unifies* all pointer targets for the pointer `root`, and concludes that the `cf` pointer "may" point to both the `cache_rbtree` and the `cycle_cf_rbtree` objects. However, because Andersen's analysis is inclusion-based, it correctly infers that the `cycle` field of the `cf` pointer can point only to the `cycle_cf_rbtree` object.

B. Constraints and Constraint Resolution Rules

In this section we discuss the various types of constraints and constraint resolution rules that are relevant to pointer analysis. Instructions that deal with pointer operations generate constraints of the four types shown below. The constraint associated with an instruction remains the same, irrespectively of whether an inclusion-style (Andersen's) pointer analysis

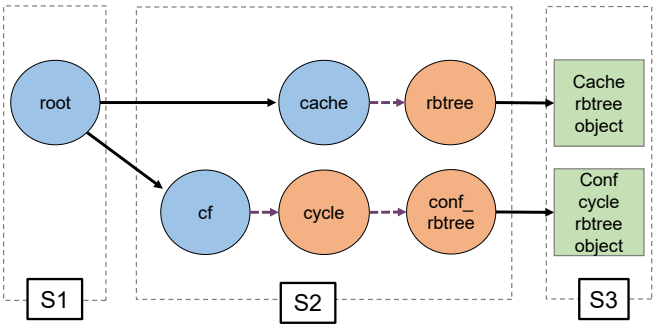

Fig. 11: Imprecision introduced by Steensgaard's Analysis. Solid arrows indicate actual points-to relationships. Dashed arrows indicate fields-of relationships. Circles indicate pointers and rectangles indicate memory objects. Steensgaard's pointer analysis forms three sets (S1, S2, S3), and derives that S1 → S2, and S2 → S3. Therefore, according to Steensgaard's pointer analysis, `cf->cycle->conf_tree` may point to both objects in set S3, and `cache->rbtree` also may point to both objects in set S3.

algorithm, or a unification-style (Steensgaard's) algorithm is used to solve them.

The constraints that are relevant to pointer analysis are:

1) $p := \&x$ (*Address-of*)
2) $p := q$ (*Copy*)
3) $p := *q$ (*Dereference*)
4) $*p := q$ (*Assign*)

Note that p and q in these examples can be single-indirection (`int *p`) or multi-indirection (`int **p`) pointers.

We assume that the relationship $pts(p)$ represents the points-to set for the pointer p. Then, the constraint resolution rules for Andersen's inclusion-style analysis are as follows:

1) $p := \&x \Rightarrow x \in pts(p)$
2) $p := q \Rightarrow pts(p) \supseteq pts(q)$
3) $p := *q \Rightarrow pts(p) \supseteq pts(pts(q))$
4) $*p := q \Rightarrow pts(pts(p)) \supseteq pts(q)$

Steensgaard's analysis is a unification-based pointer analysis algorithm. Every pointer and memory object belongs to a single "set." We assume that the operation set can be used to find the set membership of a pointer (i.e., which set a pointer belongs to). Constraints in Steensgaard's analysis are resolved by unifying these sets and we assume that the operation $join$ finds the union of two sets. The constraint resolution rules for Steensgaard's unification-style analysis are as follows:

1) $p := \&x \Rightarrow join(pts(set(p)), set(x))$
2) $p := q \Rightarrow join(pts(set(p)), pts(set(q)))$
3) $p := *q \Rightarrow join(pts(set(p)), pts(pts(set(q))))$
4) $*p := q \Rightarrow join(pts(pts(set(p))), pts(set(q)))$

C. Steensgaard Constraint Graph Representation Details

As discussed in Appendix B, different instructions generate constraints of different types. For example, LLVM instructions of type `AllocaInst`, which create and return the address

of an object, generate *Addr-of* constraints. Type casting instructions, such as `BitCastInst` and `TruncInst`, generate *Copy* constraints. `LoadInst` instructions, which dereference an IR pointer and return the value stored at the target location, generate `Deref` constraints, and `StoreInst` instructions generate `Assign` constraints.

SVF first models these instructions and their constraints as nodes on a graph, called the *pointer assignment graph* (PAG). Instruction operands that are pointers or objects are modeled as nodes in the PAG, and the IR instructions that represent the constraints are modeled as edges. SVF then clones the PAG into a *constraint graph* and begins solving the constraints. During solving, the nodes and edges of the graph are modified to reflect the constraint resolutions. While we reuse the functionality provided by SVF to build the PAG and to model calls to external functions, we use our own constraint graph implementation, which we call the *PTSGraph*.

In Steensgaard's analysis, pointers are members of points-to sets. Therefore, we need a quick way to perform set membership tests. Moreover, when the constraint solving process encounters a copy constraint, representing a statement of the form $p \coloneqq q$, where p and q are pointers, we need a way to quickly unify the two points-to sets of p and q.

In PTSGraph, we represent each points-to set with a unique identifier. A points-to set can contain multiple objects and other pointers (in case of double pointers, such as `int **p`). To ensure fast set membership tests and set unification operations, we represent set membership as a `BitVector`, a data structure provided by LLVM that is optimized for set operations. The set operations take time proportional to the size of the bit vector, and operations are performed one word at a time, instead of one bit at a time, improving performance further.

Second, points-to relationships are represented by a one-to-one, directed relationship between two sets. To improve efficiency, we represent these relationships as a `std::unordered_multi_map`. After processing each constraint in the PTSGraph, this map contains a unique mapping from one set identifier to another. However, because the intermediate processing of these constraints can occasionally result in having to store a 1:M mapping, we use an `unordered_multi_map`, to store this mapping. Because `unordered_multi_map` uses a hash table internally, lookup has an average complexity of O(k), where k is the number of set identifiers returned by the lookup operation. Because after processing of each constraint k is always 1, the lookup operation has an average complexity of O(1).

D. DFT Loop Optimizations for Array Accesses

Sensitive label lookups are significantly less expensive than cryptographic operations (our experiments in Section VI-B show that AES encryption is 430% more expensive than a label lookup), and reduce the imprecision of Steensgaard's analysis, as shown in Figure 1. However, label lookups still involve a memory read and have a non-negligible runtime overhead, especially when they are repeatedly invoked in a loop. We observed that most label lookups that occur within loops are due to byte-by-byte array traversals (either on the stack or the heap) through partially sensitive pointers—as the pointer is partially sensitive, a lookup is needed before accessing each element. Our static analysis does not distinguish between the individual elements of an array, and thus even if one element is sensitive, then all elements of the array become sensitive.

Given that these in-loop lookups incur considerable runtime overhead, we optimize them as follows. First, we use LLVM's Loop Analysis pass to retrieve all loops in the bitcode of the program. For each loop, we inspect each instruction to check if it performs a memory load or store indexed by an offset from a base pointer, of the form `v = *(ptr+i)` or `v = ptr[i]`, where `i` is the loop counter. For every such loop, we clone it and specialize the clone to unconditionally perform the required AES transformation on the identified memory operations, while the original loop body remains unchanged.

The sensitive label lookup is then *hoisted* outside the loop, and checks only the first element of the array. A conditional branch then transfers control to either the specialized or the unmodified loop, depending on the presence or absence of a sensitive label. This allows us to perform a single label lookup to ascertain the sensitivity of the entire array, instead of performing multiple byte-by-byte lookups, thus reducing the performance overhead.

E. Spectre Exploit Details

1) Spectre-PHT (Bounds Check Bypass): The Spectre-PHT PoC [40] contains a bounds check which is bypassed to leak a secret string. Listing 2 shows the bounds check in the function `victim_function`, that is speculatively bypassed to overflow `array2` and load the secret string from memory into the cache. We mark this string as sensitive and use DynPTA to harden the program. This encrypts the in-memory representation of the secret. When the secret is speculatively accessed, by overflowing `array2`, only the *encrypted* contents are loaded into the caches. Therefore, only the encrypted contents can be leaked and the confidentiality of the secret is preserved.

```
1  void victim_function(size_t x) {
2      ...
3      if (x < array1_size) {
4          temp &= array2[array1[x] * 512];
5      }
6      ...
7  }
8
9  int main(void) {
10     char* secret = "This is a secret";
11     mark_sensitive(secret);
12     ...
13 }
```

Listing 2: Code snippet for Spectre Variant-1 vulnerability

2) Spectre-BTB (Indirect Branch Poisoning): The Spectre-BTB PoC [41] contains a secret string that is leaked by redirecting the speculative execution to an appropriate gadget. Listing 3 shows the relevant snippet of code from the PoC. The function `victim_function` contains the indirect branch that can be poisoned to redirect (speculative) execution to the gadget that leaks the in-memory secret. Similarly to the

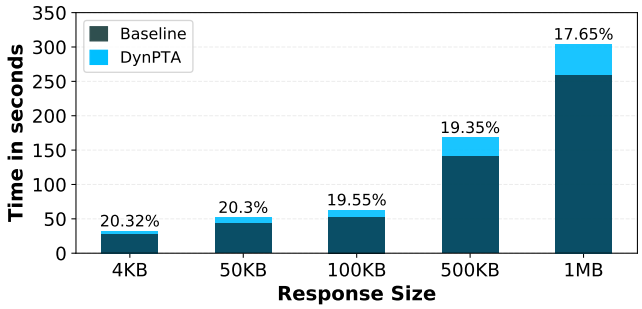

Fig. 12: Runtime overhead of Nginx for protecting passwords along with SSL private key when HTTP password authentication is enabled.

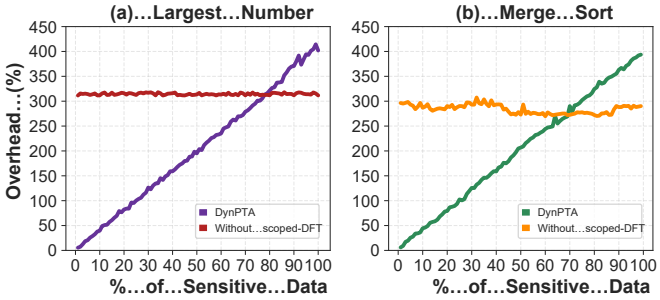

Fig. 13: Microbenchmark results of DynPTA's run-time overhead with and without scoped-DFT, for an increasing ratio of sensitive vs. non-sensitive data in the program. As the percentage of sensitive data exceeds 70–80%, the scoped DFT and label lookups become more costly than simply encrypting all objects identified by the points-to analysis.

Spectre-PHT exploit, we mark this secret as sensitive and use DynPTA to harden the program. This encrypts the in-memory representation of the secret, ensuring that only the *encrypted* contents are loaded into (and potentially leaked from) the cache, while the plaintext secret remains confidential.

```
int gadget(char *addr) {
  return channel[*addr * 1024];
}

int safe_target() {
  return 42;
}

int victim_function( *addr, int input) {
  ...
  (*addr)();
  ...
}

int main(void) {
    char* secret = "This is a secret";
    mark_sensitive(secret);
    ...
    victim(...);
}
```

Listing 3: Code snippet for Spectre Variant-2 vulnerability

F. Nginx with Password Authentication

In addition to protecting the SSL private key, we enabled HTTP password authentication for Nginx and also protect the in-memory passwords. Although this is a rarely encountered use case in real-world deployments, marking these two different types of data as sensitive results in additional instrumentation at different parts of the code. During authentication, the provided user password is checked against a list of credentials loaded in memory from a file, which we mark as sensitive. Using the same set of experiments described in Section VI-C1, we observed only a minor increase of 1% in the overall performance overhead, as shown in Figure 12. This is in line with our experience with protecting the HTTP password for Httpd and Lighttpd (Figure 9(b)–(c)).

During startup, the SSL private key is read from a file, and thus its plaintext form is briefly exposed on the stack, before being encrypted by DynPTA. In practice, these stack frames are destroyed (overwritten) right after the server's initialization completes (and the called function returns). For our work, we assume that the system starts from a clean state, and because the server has not started handling requests yet, this window of opportunity does not represent a vulnerability. Still, to illustrate that sensitive data can be protected right upon their initial introduction in memory from external sources, we marked as sensitive the stack objects in which the SSL private key is loaded temporarily during program initialization (specifically, `buf`, `data`, and `dataB` in function `PEM_read_bio`). The data in these objects is read via the `fread` Glibc call. Marking these objects as sensitive, using the `mark_sensitive` primitive, encrypts them in memory. Because `PEM_read_bio` is invoked only during program startup and these objects are never referenced again, we did not observe any performance impact due to these additional sensitive objects.

G. Microbenchmarks

To further study the performance characteristics of DynPTA as an increasing amount of application data is marked as sensitive, we implemented two microbenchmark programs and hardened them using DynPTA. The data in both programs comprise a list of 100 arrays, with each array initialized with 100,000 random integers. In each round, we can vary the percentage of arrays that are marked as sensitive. The first microbenchmark computes the largest number of all items in the list of arrays, and the second microbenchmark sorts all integers in the list of arrays using the merge sort algorithm.

For our experiments, we varied the ratio of sensitive to non-sensitive arrays in each microbenchmark and measured the run-time overhead at each point. As shown in Figure 13, as the ratio of sensitive to non-sensitive arrays increases, the overhead increases linearly as well (from 5.4% to 401% for largest number, and from 6% to 393% for merge sort).

To study the performance benefits of scoped DFT, we repeated the above experiments by disabling scoped DFT and label lookups, i.e., using the results of Steensgaard's analysis directly to encrypt *all* objects identified by the points-to analysis. As shown in Figure 13, without scoped DFT the overhead is

significantly higher and overall remains constant, irrespectively of how many arrays are actually marked as sensitive. Both microbenchmarks consist of a tight loop that reads the items from each array in the list and performs an operation on them. Because the same pointer is used to perform the indirect memory read access from each arrays, pointer analysis infers that this is a *sensitive* pointer, and thus applies the AES transformations to it. This results in *all* arrays being treated as sensitive (and requiring to be encrypted in memory), even though the programmer explicitly annotated only a fraction of them as sensitive. This results in high performance impact even though only a fraction of the arrays are marked as sensitive.

Although DynPTA performs better than this "naive" approach as long as the amount of sensitive data remains below 70–80%, scoped DFT actually becomes more costly once the amount of sensitive data exceeds this threshold. The main reason is that the cost of the excessive number of DFT label lookups at that point becomes higher than the benefit of eliding AES operations, as only a fraction of data at that point is *non-sensitive*.