TypeFlexer: Type Directed Flexible Program Partitioning

Arunkumar Bhattar [*], Liyi Li [†], Mingwei Zhu [‡], Le Chang [‡], and Aravind Machiry[*]
[*]Purdue University
[†]Iowa State University
[‡]University of Maryland, College Park

***Abstract*—Program partitioning is a proven technique for isolating potentially vulnerable code from trusted program components. We argue that an extreme isolation mechanism is not needed for all use cases. However, existing approaches tightly couple the security policy (what to partition) with the isolation mechanism (how to partition) making them inflexible. We propose TypeFlexer, which cleanly separates these concerns through a type-directed design. Our novel type system uses `tainted` annotations to mark entities that must be isolated, ensuring that tainted components do not interfere with untainted ones. To facilitate this process, we introduce TypeMatic, an automated annotation tool that not only propagates taint information according to our type rules but also identifies critical taint explosion points, allowing developers to apply explicit sanitizations where needed. We demonstrate the flexibility of our approach by designing three distinct isolation mechanisms, each with unique security guarantees and performance trade-offs. Our evaluation shows that TypeFlexer effectively contains vulnerabilities with negligible overhead as compared to the 12.8% performance penalty seen in existing state-of-the-art program partitioning techniques.**

***Index Terms*—Program Partitioning, Type Checking, Dynamic Checking, Language Based Security.**

I. Introduction

Vulnerabilities due to memory corruption in C programs remain widespread [1], [2], [3], despite many countermeasures [4]. Program partitioning has long been explored for security: it separates potentially vulnerable components from the rest [5], [6], [7], [8], [9]. However, these techniques often impose heavy overhead by combining partitioning with strict Software Fault Isolation (SFI) [10]. Specifically, in addition to partitioning, existing techniques also provide SFI such that code in the untrusted region cannot affect code in the trusted region, *e.g.,* by executing untrusted partition in a separate process [9] or executing in a Trusted Execution environment [8]. However, this extreme partitioning is not always necessary for certain functions and also infeasible in certain resource-constrained environments. For instance, consider a string processing function that scans for a pattern. Having a complete SFI for such a function adds a lot of overhead and may not be necessary. Ideally, we want to isolate this only against spatial safety violations (*e.g.,* buffer overread), as the function cannot contain other issues (*e.g.,* use-after-free). But existing partitioning techniques do not allow this as they are specialized for a specific SFI technique (Section II).

In other words, existing techniques combine policy (*i.e., what* to partition) with mechanism (*i.e., how* to partition). We argue that existing general-purpose program partitioning mechanisms are restrictive and inflexible for different use cases (*e.g.,* isolation from spatial safety vulnerabilities). We need a technique that enables developers to flexibly partition programs and enforce isolation – balancing safety guarantees and performance overhead.

In this paper, we present TypeFlexer, a Type-Directed Flexible Program Partitioning system. The main design principle of this system is to separate partitioning (*i.e.,* policy) from enforcing the separation between partitions (*i.e.,* mechanism). Once a program is partitioned, users can choose enforcement mechanisms based on the desired guarantees and tolerable performance overhead. For instance, for the same program, the user may use complete SFI for desktop systems v/s isolation against only spatial safety issues on embedded systems.

We propose a type directed partitioning through untrusted types. Specifically, we introduce **tainted** qualifiers that can be used to annotate pointer types (`_TPtr<T>`) or functions (`_Tainted`) . The tainted types form the untrusted partition or `u` *region* (can only access tainted types), and the non-tainted types will be the trusted partition or `c` *region* (has complete access). Through static type checking, our type system ensures that tainted pointers cannot directly affect untainted pointers. After successful type checking, TypeFlexer creates two sets of source files: `u` region and `c` region, containing tainted and untainted functions, respectively.

To assist developers, we present TypeMatic, an automated tool that propagates `tainted` annotations from a minimal set of developer-annotated pointers (seeds), thereby preventing *taint explosion* due to over-annotation. We formally prove a *clean separation theorem*, ensuring that in a well-typed system, tainted types remain strictly isolated from the rest of the program. This guarantee prevents tainted pointers from undermining the program's integrity or causing issues within the `c region`.

After partitioning, developers can choose any of our three Selective Software Fault Isolation (SSeFI)-inspired mechanisms (e.g., minimal spatial checks vs. full SFI). Each mechanism incurs different runtime overheads while providing varying degrees of isolation. Our approach also integrates with alternative backends (e.g., compiling `u region` with WASM [11]). We also design several optimizations for common cases to reduce the performance overhead of our isolation mechanisms.

In summary, the following are our contributions:

- We design a type system ensuring `u` cannot affect `c`, formally proving a *clean separation theorem*.
- We build TYPEMATIC and a partitioner (CHECKMATE) to automate annotation and emit separate source sets.
- We develop three isolation modes with different guarantees/overheads.
- Our evaluation shows 0%–43.1% overhead on real benchmarks and that TYPEFLEXER can successfully isolate 16 real-world vulnerabilities. Our partitioning mechanisms offer the same extent of isolation compared to a closely related work PTRSPLIT at a significantly lesser overhead.

II. BACKGROUND AND MOTIVATION

This section presents the necessary background and motivation for our work.

Memory Safety. Memory safety ensures all program memory accesses target valid objects [12]. Violations include: *Spatial violations* (e.g., out-of-bounds reads) and *Temporal violations* (e.g., use-after-free).

Program Partitioning. Program partitioning [13] splits code into isolated parts. Many solutions [5], [6], [7], [8], [9] adopt a *data-centric* model [8], [9], grouping functions accessing sensitive data into a trusted partition. Existing techniques primarily employ program analysis and dynamic instrumentation to enable transparent communication between functions in trusted and untrusted partitions. Because they focus on inter-partition interactions, these techniques are specialized for specific isolation mechanisms. Some, such as GLAMDRING [8], rely on special hardware support. Other software-based approaches also tailor their partitioning to the chosen isolation mechanism. For example, PTRSPLIT [9] assumes that partitions run as separate processes, exchanging data through marshaling. Its partitioning mechanism (via Program Dependence Graph (PDG)) is specialized for marshaling, making it difficult to support other mechanisms like sandboxing. DYNPTA [14] tries to reduce the overhead by using static points-to analysis and avoiding unnecessary checks. However, similar to PTRSPLIT, DYNPTA also combines both policy and mechanism. HAKC [15] focuses on kernel modules, whereas CHECKED C [16] ensures spatial safety but lacks flexible isolation. Table I summarizes drawbacks of notable systems.

Need for Selective Fault Isolation. Existing partitioning techniques focus on protecting sensitive data, enforcing complete Software Fault Isolation (SFI) so that all executions, data, and faults within one partition do not affect others. This strict isolation yields high overheads (37%–163%) [8], [9], and some mechanisms (*e.g.,* Intel SGX) may not be available on all systems. Such heavyweight isolation is often unnecessary for practical security. A recent Microsoft study [2] reports that $\approx 70\%$ of the CVEs patched over the past decade were memory-safety bugs (both spatial and temporal). Other works [18], [19], [20], [21] reveal that certain functions are especially prone to specific vulnerabilities, while others cannot cause temporal issues (such as use-after-free) if they do not invoke (*e.g.,* directly or transitively) any heap operations (*e.g.,* `malloc`/`free`). Hence, isolating such functions solely against spatial violations can be more efficient without compromising security.

We need a program partitioning technique that flexibly supports different isolation mechanisms for different partitions.

III. OVERVIEW

Figure 2 shows the interaction between various components of TYPEFLEXER from a developer perspective. The developer first annotates the source program and then specifies an isolation mechanism. TYPEFLEXER then creates an executable such that the trusted (`c` region) partition is sufficiently isolated from the tainted (`u` region) partition.

We use tainted (`u` region) to denote untrusted code or data, *i.e.,* that needs to be partitioned out and should be isolated from the rest of the program. Similarly, we use untainted (`c` region) to mean trusted code or data. TYPEFLEXER's goal is to isolate vulnerabilities that originate in code or data explicitly annotated as tainted (`u` region). The trusted partition (`c` region) is assumed free of memory-safety bugs, and TYPEFLEXER protects it from `u` region. An adversary can fully control inputs to `u` region code and exploit arbitrary memory-safety flaws there to corrupt other `u` region objects, but must not be able to affect `c` region.

A. Running Example

We use the program in Figure 1 as the running example to explain the developer workflow in using TYPEFLEXER. Figure 1 shows the C code of the redacted version of a simple network server with `server_loop` as its entry point (⏩). The server runs in a loop and calls `handle_request`, which handles a network request. The function `handle_request` reads data from the socket, and (based on the first byte), `process_req1` is called to handle the request.

The Vulnerability. There are arbitrary memory write vulnerabilities (indicated by 🐞s) in `process_req1`. This is because the `for` loop only checks if `cp2` has at least one more byte (*i.e.,* `(cp2-tmp)< (CMD_S -1)`). But if the character is `'<'`, four bytes are written, resulting in a 2-byte buffer overwrite and an arbitrary write as the last write uses `i` as an index.

Goal. Given the complexity of the `process_req1` function, which also processes network data, a developer might want to partition the function from the rest of the program.

B. Automated Partioning

First, developer annotates the function `process_req1` as `_Tainted` indicated by ①. Second, the code with the initial annotation is provided to TYPEMATIC, which will propagate the annotations to all the other interacting pointers based on the TYPEFLEXER Type System, as indicated by ②. Finally, this well-typed annotated program will serve as input for our system. Our type checker ensures that annotations follow the typing rules, and accepts program that follow all the type rules.

Flexible Typing Rules. We *allow* tainted pointers (_TPtr) to appear alongside ordinary C pointers within `c` region source code, significantly enhancing programmability. Since

TABLE I: **Comparison of Software-Based Partitioning and Isolation Techniques.**

Feature	**TypeFlexer (Our Work)**	**HAKC** [15]	**DataShield** [17]	**DynPTA** [14]	**Checked-C** [16]	**PtrSplit** [9]
Type-Based Isolation	✓	✓	×	×	✓	✓
Back-end Agnostic	✓	×	×	×	✓	✓
Flexible Isolation	✓	×	×	×	×	×
Balanced Perf & Security	✓	✓	×	✓	✓	×
Minimal Annotations	✓	✓	×	×	×	✓
Taint Explosion Analysis	✓	×	×	×	×	×

```
⏩ // Entry point
void server_loop(int sock_fd) {
 unsigned b_z;
 struct queue *p;
 ...
 setup(crypt_key);
 ...
 while(1) {
  ...
  p = malloc(
  sizeof(struct queue) * b_z);
  ...
  if (handle_request(sock_fd)) {...}
 }
}

int handle_request(int sock_fd) {
 char buff[MAX_MSG_SIZE];
③ ⚙👤_TPtr<char> buff;

 int rc = -1;
 ssize_t r_len;
 char filename[MAX_LEN];
 r_len = read(sock_fd, buff,
              MAX_MSG_SIZE);
 if (r_len > 0 &&
     r_len < MAX_MSG_SIZE) {
  switch(buff[0]) {
   case REQ1:
    rc = process_req1(buff, r_len);
    break;
   case REQ2:
        int* digit_loc;
        ...
  }
 }
 return rc;
}
```

```
int process_req1 ① 👤_Tainted int (
char *msg ② ⚙_TPtr<char> msg ,
                 size_t m_l) {
 int rc = -1, i;
 if (m_l > MIN_SIZE) {
  msg += sscanf(msg, "%d", &i);
  if (i > 0) {
    char *cp1 ② ⚙_TPtr<char> cp1;
    char *cp2 ② ⚙_TPtr<char> cp2;
    char tmp[CMD_S] ② ⚙_TPtr<char> tmp;

    for ( cp1 = msg, cp2 = tmp;
        *cp1 != '\0' &&
        cp2 - tmp < CMD_S - 1;
        ++cp1, ++cp2 )
    {
     switch ( *cp1 )
     {
       case '<':
        // We write 3 characters into
        // cp2 but we only checked for 1
        // charecter.
        *cp2++ = '&';
        🐞 *cp2++ = 'l';
        🐞 *cp2++ = 't';
        // Arbitrary write vulnerability.
        🐞 *(cp2 + i) = ';';
      break;
        ...
     } } }
 }
 return rc;
}
```

Fig. 1: C program snippet of a simple network server with an arbitrary memory write vulnerability indicated by 🐞. The circle markers show type annotations made by the developer (👤) and automatically propagated (⚙) during partitioning the function `process_req1` from the rest of the code.

our threat model assumes the `c` region to be memory safe, it does *not* attempt to protect the `u` region from misbehaving `c` region code. However, the flow of data between tainted and untainted pointers in `c` region is governed by the formal mode-typed rules detailed in Section IV, enforced interactively by TYPEFLEXER's type-checker.

Moreover, arbitrary pointer casts that include common patterns, such as `void*`, integer intermediates, and unions, are fully permitted within `u` region code. Only casts that leak trusted (`c` region) pointers into untrusted code are rejected. Even in scenarios involving impractical pointer annotations (taint explosion), developers always retain the flexibility to explicitly marshal (sanitize) data from tainted (`u` region) to untainted (`c` region) pointers, effectively breaking taint propagation and ensuring practical applicability with moderate developer intervention.

C. Selecting the Desired Isolation Mechanism

Instead of aiming for complete isolation, our approach focuses on Selective Software Fault Isolation (SSeFI). Formally, a partition is defined as ϕ-SSeFI if any violation of the property ϕ is confined within that partition and does not impact other partitions. For instance, achieving spatial safety-SSeFI in a partition ensures that spatial safety violations are isolated within that specific partition.

Developers need to choose a suitable SSeFI mechanism (Section V) to clearly separate tainted and untainted regions. For isolation mechanisms (ISOMEM (or I_{mem}) and SANDMEM (or S_{mem})) — The annotated code is compiled with the TYPEFLEXER compiler to produce an executable with the designated security feature.

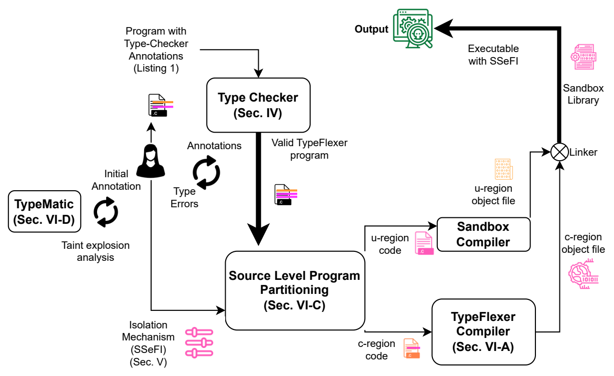


Fig. 2: Overview of interaction between various phases of TYPEFLEXER.

In contrast, SANDBOX (or S_{all}) isolates both untrusted data and code into `u` region. Consequently, untrusted functions must be moved to separate `u` region source files, allowing compilation with a sandbox-specific compiler for fully isolated runtime behavior. To facilitate this process, our source-level partitioner, CHECKMATE, divides the codebase into two categories:

- *Untainted* or *Trusted* partition (`c` region short for *clean/checked* region), which contains source files referencing both tainted and untainted pointers, as well as function calls to tainted functions;
- *Tainted* or *Untrusted* partition (`u` region short for *unclean/unchecked* region), which contains source files defining tainted functions.

The tainted partition is compiled with our TYPEFLEXER compiler to enforce strict isolation, while the untainted partition can be compiled with any sandbox-specific compiler. Finally, object files from both partitions are linked with the required helper libraries to produce the final executable.

Summary. We define taint annotations that can be added to any function, `struct`, or pointer type declaration, *e.g.,* struct field, struct pointer, function pointer, etc. Our type checker statically ensures that the tainted types do not interact unsafely with untainted types. At runtime, the tainted types are checked to ensure that they indeed point to the tainted region. The exact check depends on the selected isolation mechanism. For instance, if a struct field is declared as tainted, then all dereferences to the corresponding member (irrespective of the object) will be considered as tainted dereferences and will be checked. Listing 1 illustrates our static type checking and insertion of dynamic checks.

Out-of-Scope Attacks. TYPEFLEXER does not defend against micro-architectural and side-channel leakage (Spectre-class transient execution, cache probing, etc.). These threats require orthogonal defenses such as input sanitization, constant-time coding, or hardware mitigations. Making these scoping considerations explicit allows TYPEFLEXER to keep dynamic checks lightweight while still providing strong integrity guarantees for trusted code.

```
struct foo {
 _TPtr<int> ptr;
 char *ptr2;
};
struct foo *obj1;
int b;
_TPtr<int> a1;
int *a2;

// Here, we add a check to ensure
// that ptr points to a tainted memory region.
.....obj1->ptr[0];......✅🟨

// Here, we do not add a check as
// the field is not tainted.
......obj1->ptr2[x];......✅🟨

// No check will be added in the following cases as
// there are no deferences.

// This is Okay, as propagation happens
// from tainted to tainted.
a1 = obj1->ptr + b; ✅

// This will be rejected (i.e., not allowed)
// by the type checker.
a2 = obj1->ptr + b; 🔴
```

Listing 1: Example illustrating TYPEFLEXER's static type checking (OK (✅)/NOTOK (🔴)) and dynamic checking (🟨).

$$
\begin{array}{llcl}
\text{Variables: } x & \multicolumn{3}{l}{\text{Integers: } n ::= \mathbb{Z}} \\[4pt]
\text{Mode:} & m & ::= & \texttt{c} \mid \texttt{u} \\
\text{Type:} & \tau & ::= & \texttt{int} \mid \texttt{ptr}^m\ \tau \\
\text{Expression:} & e & ::= & n{:}\tau \mid x \mid e + e \mid (\tau)e \\
 & & \mid & \texttt{malloc}(m,\omega) \mid \texttt{free}(e) \mid \\
 & & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ret}(x, n{:}\tau, e) \\
 & & \mid & \texttt{ret}(x, n{:}\tau, e) \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e \\
 & & \mid & *\ e \mid *\ e = e \mid \texttt{to}(m)(\overline{x})\{e\}
\end{array}
$$

Fig. 3: COREFLEXER Syntax

IV. USING TYPING TO DISTINGUISH POINTERS

The partitioning is essentially driven by our type system that enforces the safe usage of tainted types. Our type system enforces the policy of clear separation between tainted and untainted types.

A. Formalism Overview

We use a simple language called COREFLEXER to explain our typing rules. Figure 3 shows the syntax of COREFLEXER, which is based on a subset of the Compcert formalism [22], with some features coming from the Checked C formalism [23]. We classify types as integers or pointers. Every pointer type ($\texttt{ptr}^m\ \tau$) includes a mode annotation that is either tainted (`u`) and non-tainted (`c`), and a content type (τ) denoting the valid type it points to. `u` mode pointers are those living in `u` region, such as all pointers in `process_req1` in Figure 1, while `c` pointers are the pointers that are not marked `tainted` in `handle_request` (Ex: `filename`).

Disallowing Unsafe Types. The well-formedness rules of our type system prevent unsafe types from being constructed.

Consider the type $\texttt{ptr}^{\texttt{u}}\ (\texttt{ptr}^{\texttt{c}}\ \texttt{int})$, which describes a tainted pointer to a `c` mode pointer. This is not well-formed in COREFLEXER because it potentially exposes the `c` mode pointer addresses in a `u` region when the tainted (`u`) pointer is used. Nevertheless, we can have a `c` mode pointer whose element is a tainted pointer, *e.g.,* $\texttt{ptr}^{\texttt{c}}$ ($\texttt{ptr}^{\texttt{u}}$ `int`) is a valid type. This is the formalism to enforce the property that there are no untainted pointers in `u` region heap in Section V-A1.

Type System. Our type system is both flow-sensitive and gradually-typed. It introduces additional static checks during the type-checking stage, which are then verified in the semantic evaluation stage. The type-checker enforces constraints on the use of `u` pointer types to prevent their operation within the `u` regions from impacting the execution in the `c` regions. This enforcement is guided by a set of well-defined typing rules.

Figure 4 shows a few of our typing rules. Each typing rule has the form $\Gamma \vdash_m e : \tau$, which states that in a type environment Γ (mapping variables to their types), expression e will have type τ if evaluated in context mode m, indicating that the code is in m region.

The type rule (T-DEF) ensures that pointers are used with the right modes in the right region ($m' \leq m$). For example, `process_req1` in Figure 1, which is in `u` region, should only contain `u` pointers. For pointer casting, rule T-CASTPTRU permits random casting for `u` region, with the only requirement that one cannot cast a `u` mode pointer to `c` mode. For `c` region casting, we only permit the casting from pointers to `int` (T-CASTPTRC and T-CASTINT).

Similarly, our other rules ensure that there are no unsafe interactions between tainted (`u`) and non-tainted (`c`) pointers. Any violations of these rules will be displayed to users. The violations also provide information on how to fix them, enabling users to add annotations until the type-checker accepts them interactively.

Our typing rules statically ensure safe interaction between `c` and `u` region pointers — providing the partitioning policy. However, ensuring that the `u` region pointers do not access `c` region memory (*i.e.,* isolation mechanism) cannot be done statically and will be performed through dynamic checks, depending on the target isolation mechanism (Section V).

B. Meta Theories

Here, we discuss our main meta-theoretic results for CORE-FLEXER: non-exposure, type preservation, and clean separation. These proofs have been conducted in our Coq model.

We first show the non-exposure theorem, where code in `u` region cannot access a valid `c` pointer address. By accessing, we refer to the dereference, assignment, `malloc`, and `free` operations.

Theorem 1 (Non-Exposure): For any COREFLEXER program e, heap $\mathcal{H}$, stack φ, type environment Γ, and variable predicate set Θ that are all are well-formed and well typed ($\Gamma \vdash_m e : \tau$ for some τ), if there exists φ', Θ', $\mathcal{H}'$ and e', such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{\texttt{u}} (\varphi', \Theta', \mathcal{H}', e')$ and $e = E[e']$ and $mode(E) = \texttt{u}$, thus, e' does not access a `c` pointer.

The non-exposure theorem prevents two vulnerabilities. First, it prevents the misuse of pointers in `u` region from affecting the `c` region execution, such as 🐞 in Figure 1. .

Theorem 2 (Type Preservation): For any COREFLEXER program e, heap $\mathcal{H}$, stack φ, type environment Γ being well-formed and consistent ($\Gamma \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma \vdash_{\texttt{c}} e : \tau$ for some τ), if there exists φ', Θ' $\mathcal{H}'$ and e', such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_m (\varphi', \Theta', \mathcal{H}', e')$, then $\mathcal{H}'$ is consistent with $\mathcal{H}$ ($\mathcal{H} \triangleright \mathcal{H}'$) and there exists Γ' and τ' that are well formed, consistent ($\Gamma' \vdash \varphi'$ and $\mathcal{H}' \vdash \varphi'$) and well typed ($\Gamma' \vdash_{\texttt{c}} e : \tau$).

We define a state to be *stuck* and *critically stuck* below, and then show our main result, *clean separation*, which suggests that a well-typed program can never be critically stuck in `c` code regions.

Definition 1 (Critically Stuck): For a program e and environment tuple $(\varphi, \Theta, \mathcal{H}, e)$, we define it to be stuck as that there is no transition tuple $(\varphi', \Theta', \mathcal{H}', r)$, such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{\texttt{c}} (\varphi', \Theta', \mathcal{H}', r)$; it is critically stuck, when $(\varphi, \Theta, \mathcal{H}, e)$ is stuck, and e is of the three situations:

- $e = *\ n{:}\texttt{ptr}^{\texttt{u}}\ \tau$.
- $e = *\ n{:}\texttt{ptr}^{\texttt{u}}\ \tau = n'{:}\tau'$.

Theorem 3 (Clean Separation): For any COREFLEXER program e, heap $\mathcal{H}$, stack φ, type environment Γ, and set Θ that are well-formed and consistent ($\Gamma \vdash \varphi$ and $\mathcal{H} \vdash \varphi$), if e is type-preserved ($\varphi \vdash_{\texttt{c}} e : \tau$ for some τ) and there exists φ_i, Θ_i, $\mathcal{H}_i$, e_i, and m_i for $i \in [1, k]$, such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{m_1} (\varphi_1, \Theta_1, \mathcal{H}_1, e_1) \longrightarrow_{m_2} \ldots \longrightarrow_{m_k} (\varphi_k, \Theta_k, \mathcal{H}_k, r)$, then r can never be *critically stuck*.

Clean separation suggests that `c` and `u` regions are completely separated, because the tainted pointers, the only types of pointers that communicate the `c` and `u` regions, do not cause any problem in `c` regions. The 🐞 in Figure 1 is dynamically caught through the dynamic checks in TYPEFLEXER and it is included in the non-exposure theorem. The additional guarantee the clean separation provides is to ensure that even if a `u` mode pointer is freed in `u` region, when we access it in `c` region, our compiler can discover the error, explained in Section V-A as the use-after-free dynamic check. We provide more details of our formalism in Appendix H.

V. SELECTIVE SOFTWARE FAULT ISOLATION (SSEFI)

This section presents three distinct fault isolation mechanisms, each offering varying guarantees and overhead. These mechanisms differ in how they isolate the tainted region (`u` region) from the untainted region (`c` region). Table II provides a summary of each isolation mechanism, detailing their corresponding security guarantees, pros/cons, and example use cases, offering guidance to developers on selecting the appropriate sandbox for their needs.

A. ISOMEM (I_{mem})

I_{mem} enforces spatial and temporal SSeFI based on tainted pointers, preventing buffer overflows or use-after-free in `u` from corrupting `c` pointers.

For the example in Listing 2, there is an index out-of-bounds access through `buff` at line 5 and use-after-free of `p` at line 11. Even with these bugs, I_{mem} protects `ptr` or other untainted pointers by:

$$\textsc{T-Def}\ \frac{m' \leq m \qquad \Gamma \vdash_m e : \mathtt{ptr}^{m'}\ \tau}{\Gamma \vdash_m *\ e : \tau} \qquad \textsc{T-CastPtrC}\ \frac{\Gamma \vdash_{\mathtt{c}} e : \tau}{\Gamma \vdash_{\mathtt{c}} (\tau)e : \tau} \qquad \textsc{T-CastInt}\ \frac{\Gamma \vdash_{\mathtt{c}} e : \tau}{\Gamma \vdash_{\mathtt{c}} (\mathtt{int})e : \mathtt{int}} \qquad \textsc{T-CastPtrU}\ \frac{\Gamma \vdash_{\mathtt{u}} e : \tau'}{\Gamma \vdash_{\mathtt{u}} (\mathtt{ptr}^{\mathtt{u}}\ \tau)e : \mathtt{ptr}^{\mathtt{u}}\ \tau}$$

Fig. 4: Selected typing rules.

SeFI Mechanism	`u` region (tainted) Pointers	`u` region (tainted) Code	`c` region (Code & Data)	Security Guarantees	Pros and Cons	Example

I_{mem}	Hoard	`c` region	Regular C	Spatial & Temporal Isolation	(–) Complex memory tracking (interval trees);
					(–) Slow T_Check;
					(+) Flexible, diverse allocations;
					(+) Supports custom allocators.
						Pointers needing frequent resizing

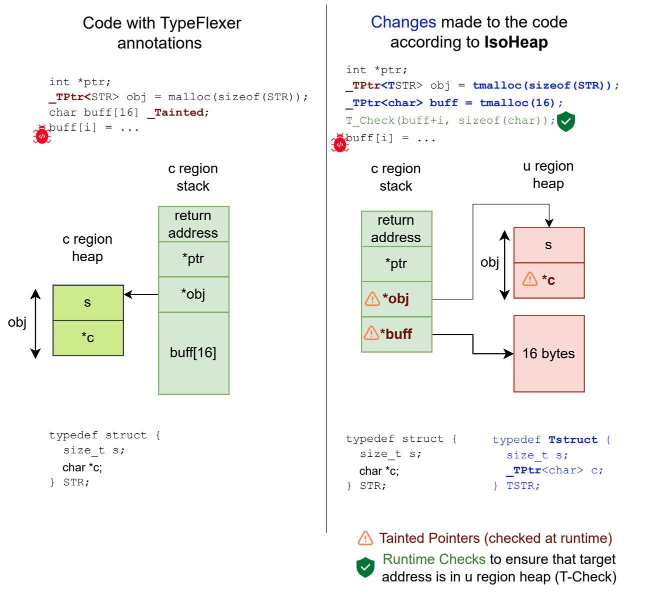
S_{mem}	Sandbox			Spatial & Temporal Isolation	(+) Simple tracking (base–upper bound);
					(+) Fast T_Check;
					(+) Integrates with S_{all};
					(–) Linear memory model.
						Deterministic-time pointers or full isolation needs

S_{all}	Sandbox	Sandbox		Complete Isolation	(+) Strong isolation from non-pointer vulns; (–) Performance overhead.	Functions demanding maximum isolation

TABLE II: Summary of SSeFI Mechanisms.

```
char *ptr;
...
char buff[10] __Tainted;
...
🐞 buff[i] = 'A'; // OOB if i>10
...
_TPtr<int> *p = tmalloc(...);
...
free(p);
...
🐞 *p = ...;      // use-after-free
```

Listing 2: Example illustrating SSeFI Mechanisms.


Code with TypeFlexer annotations
Changes made to the code according to IsoHeap
c region stack
c region heap
obj
s
*c
return address
*ptr
*obj
buff[16]
c region stack
u region heap
return address
*ptr
*obj
*buff
obj
s
*c
16 bytes
Tainted Pointers (checked at runtime)
Runtime Checks to ensure that target address is in u region heap (T-Check)

```
int *ptr;
_TPtr<STR> obj = malloc(sizeof(STR));
char buff[16] _Tainted;
buff[i] = ...

typedef struct {
  size_t s;
  char *c;
} STR;
```

```
int *ptr;
_TPtr<TSTR> obj = tmalloc(sizeof(STR));
_TPtr<char> buff = tmalloc(16);
T_Check(buff+i, sizeof(char));
buff[i] = ...

typedef struct {          typedef Tstruct {
  size_t s;                 size_t s;
  char *c;                  _TPtr<char> c;
} STR;                    } TSTR;
```

Fig. 5: I_{mem}'s Selective Fault Isolation (SeFI) mechanism.

- *Dedicated `u` region Heap:* All tainted buffers are allocated separately (tainted heap) via `tmalloc`. The left snippet in Figure 5 shows tainted pointer annotations; the right shows how local tainted buffers (e.g., `buff`) become heap allocations.
- *Dynamic Checks (`T_Check`):* During CodeGen, the TYPEFLEXER compiler wraps *every operation whose evaluation may dereference from a tainted pointer* (*e.g.,* `*p`, `p[i]`, `p`→`f`) with an inline guard `T_Check(addr,sz)`. TYPEFLEXER's type-checker (section IV) drives insertion, so untainted accesses are never guarded. At runtime, `T_Check` validates that the byte range $[addr, addr{+}sz)$ lies within the active `u` region heap segment(s). For I_{mem}, we maintain a segment tree keyed by `tmalloc`/`tfree` metadata. This metadata is stored in the `c` region such that `u` region code cannot corrupt this metadata through arbitrary memory access from within the `u` region.
  The guard does *not* inspect pointee contents; payload sanitization is the responsibility of interface code (Section III). We further reduce dynamic overhead with Loop Sanity-check Code Motion (Section VI-B), which hoists invariant checks out of hot loops.

`u` region Heap. We adapt Hoard [24] for the tainted heap, which avoids inline metadata, thus preventing temporal violations (*e.g.,* use-after-free) from exploiting heap operations [25]. Similar to any allocator, Hoard provides a level of indirection. It uses `mmap` to get chunks (spanning multiple pages) from OS and then manages these chunks for the application needs. Subsequently, Hoard releases memory using `munmap`. We keep track of valid chunks by instrumenting at `mmap` and `munmap` call sites. We store valid intervals using a modified segment tree [26] with a small cache for quick lookups. As shown in Section VII, this optimization makes I_{mem} nearly overhead-free.

1) Security Analysis: Here we present the security analysis and discuss how I_{mem} prevents various attacks.

Protecting the `c` region stack and return addresses.. In

I_{mem}, as illustrated in Figure 5, the runtime stack is shared by both `c` and `u` regions. However, no tainted buffers reside on the stack. Instead, as shown in the upper right of Figure 5, developers convert all local buffers (e.g., `buff`) to tainted heap buffers allocated via `tmalloc` in the `u` region heap. Every access through tainted pointers (e.g., `ptr`, `obj`) is guarded by a runtime check (`T_Check`) to ensure that the target address lies in the `u` region heap. Consequently, although these tainted pointer variables reside on the stack, they cannot overwrite stack variables, including return addresses.

Protection against inter-object spatial safety violations.. As discussed in Section V-A, `T_Check` only confirms whether a pointer targets the `u` region heap; hence, it cannot prevent spatial violations *within* `u` region. For instance, on the right side of Figure 5, an out-of-bounds write through `buff` could overwrite other `u` region objects (e.g., `*c`), yet it would not affect any `c` region (untainted) pointers. We enforce this isolation by ensuring that *No untainted pointers in the* `u` *region heap:* Per the well-formedness rules any pointer nested within a tainted pointer must itself be tainted.

For example, `int **` cannot be expressed as `_TPtr<int*>`, because the outer pointer would encapsulate an untainted pointer.

Handling composite types: These rules also apply to composite types (e.g., `structs`). On the left side of Figure 5, the pointer `obj`, referencing `STR`, is tainted. On the right side, developers redefine `STR` to `TSTR` (i.e., a `Tstruct`), replacing all internal pointers (e.g., `c`) with tainted types. Since TYPEFLEXER disallows `_TPtr<struct str>`, all declarations referencing `STR` must be updated accordingly.

At compile time, `Tstruct` types are enforced to contain only tainted pointers or nested `Tstructs`; other fields remain unchanged. This ensures no unchecked addresses can leak through tainted structures. Finally, the type system flags any violation at compile time and inserts instrumentation solely for tainted pointers.

```
_TPtr<TSTR> obj = ...
...
// T_Check(obj, sizeof(obj))
// T_Check(obj->c, i*sizeof(char))
...obj->c[i] =
...
STR *obj1 = ...
...
// no runtime checks as its
// an untainted field
...obj1->c[i]..
```

Listing 3: Example illustrating handling of composite types.

For instance, consider the example in Listing 3: Here both `obj` and `obj1` are pointers to types of the same size. However, that `obj` is of type `TSTR`, which is the same as `STR`, except that all pointer fields are declared tainted (right side of Figure 5). This enables our type-checker to distinguish between `obj`$\rightarrow$`c` (a tainted pointer field) and `obj1`$\rightarrow$`c` (untainted pointer field) differently and add the runtime checked (`T_Check`) to only tainted pointer accesses. In summary, given that all pointers in `u` region heap are tainted pointers, any inter-object (or intra-`u` region) overflows can only affect tainted pointers. In addition, our `T_Check` ensures that all accesses through tainted pointers are to `u` region heap. This shows that *inter-object overflows in* `u` *region cannot affect untainted pointers.*

2) Overhead Aspects $T(I_{mem})$: I_{mem} overhead comprises: *(i)* extra heap allocations $T(\Omega)$, *(ii)* pointer checks $T(\Theta)$. Hence:

$$T(I_{mem}) = T(\Omega) + T(\Theta). \tag{1}$$

B. SANDMEM *(S_{mem})*

This mechanism closely resembles ISOMEM's approach except for the `u` region heap and the corresponding `T_Check` implementation.

In S_{mem}, a sandbox (e.g., WASM) manages the `u` region heap. Therefore, `tmalloc` (and consequently, `tfree`) call sandbox-specific routines. Unlike I_{mem}, which stores the `u` region heap in the same address space, S_{mem} places it in a separate address space.

A key benefit of sandboxing is a simpler `T_Check`: sandboxes typically define static lower and upper bounds, so `T_Check` merely confirms that a pointer falls within this range. In contrast, I_{mem} may require a tree traversal for each tainted memory access (see Section VII), which can be more costly in certain cases.

Sandboxes also employ opaque identifiers to represent pointers, which necessitates pointer swizzling, *i.e.,* translating opaque identifiers into usable addresses. This adds complexity and overhead, as additional instrumentation is needed to convert the opaque values before passing them to library calls.

However, using opaque identifiers also hinders attacks that rely on inter-object spatial violations. To overwrite a `u` region pointer with a desired address, an attacker now needs the sandbox-specific opaque value of that address—a requirement that imposes additional barriers compared to I_{mem}, where attackers can directly use raw address values.

1) Overhead Aspects $T(S_{mem})$: The overhead of S_{mem} is similar to that of I_{mem} (see Eq. 1), except that sandbox allocations replace the in-process heap allocations, and pointer swizzling adds an extra overhead component. Formally:

$$T(S_{mem}) = T(\sigma) + T(\Theta) + T(\beta) \tag{2}$$

SandMem keeps both `c` region and `u` region within a single process, consequently calls between `c` region and `u` region functions are ordinary in-process function jumps, incurring no additional IPC overhead. The main additional cost component is due to pointer-swizzling, which is captured by the $T(\beta)$ term.

C. SANDBOX *or Complete SFI (S_{all})*

This approach implements a conventional SFI (Software Fault Isolation) model by fully isolating tainted code and data from their untainted counterparts. Although mechanisms like I_{mem} and S_{mem} provide partial isolation, they still run tainted and untainted functions in the same address space with the same privileges—leaving them open to data-oriented attacks (e.g., via inter-object buffer overflows [27]) and illicit system calls. For security-critical contexts, a complete SFI setup is therefore warranted.

1) Complete SFI via S_{all}: S_{all} compiles tainted functions and data into a sandbox environment (e.g., WASM) that is accessible only through dedicated sandbox APIs. Manually modifying every call site in the untainted region to redirect to sandboxed code would be cumbersome and would require precise pointer analysis [28]. Instead, we implement an *indirection strategy*, where each tainted function's body is replaced with a call to its sandboxed variant. For example:

```
_Tainted int process_req1(_TPtr<char> msg, size_t m_l) {
+ return w2c_process_req1(_SBX_(), msg, m_l);
}
```

All invocations of `process_req1` (including indirect calls via function pointers) are thus routed to the sandboxed function. The original logic of `process_req1` is moved into a new function, `w2c_process_req1`, which is compiled by a sandbox-specific toolchain. Our source-rewriting tool (CHECKMATE) automates these transformations for well-typed programs.

Within the `u` region (sandbox), explicit `T_Check` calls are unnecessary because the WASM runtime already traps out-of-bounds memory accesses. In the untainted region, however, each tainted pointer (e.g., a buffer in `handle_request`) remains guarded by `T_Check`, to prevent confused-deputy attacks [29], [30].

Pointer swizzling is needed at the sandbox boundary since WASM uses 32-bit pointers, whereas native code uses 64-bit pointers. For instance, a structure field may occupy 4 bytes when tainted but 8 bytes when untainted, causing misaligned offsets in subsequent fields. The type-checker addresses this by marking such structures as `Tstruct` and automatically performing pointer swizzling when accessed.

2) Overhead Aspects $T(S_{all})$: Like S_{mem}, S_{all} incurs overhead for sandbox-based allocations and pointer swizzling, plus an additional cost ($T(\phi)$) for running code inside the sandbox. This also includes memory allocation time ($T(\sigma)$) from Eq. 2. Since sandboxes enforce numerous runtime checks, executing code within a sandbox is generally slower than native execution. Formally:

$$T(S_{all}) = T(\sigma) + T(\Theta) + T(\beta) + T(\phi) \tag{3}$$

3) Handling Callbacks: Tainted functions normally only invoke other tainted functions. However, there may be cases where a tainted function (e.g., an input-processing routine) needs to call back into an untainted password-check function. TYPEFLEXER supports these scenarios via function-pointer callbacks (Section VI-C), maintaining isolation while allowing necessary cross-boundary function calls.

VI. IMPLEMENTATION

As outlined in Section III, TYPEFLEXER comprises three primary components: (1) a compiler (type-checker), (2) a source-level program partitioner (CHECKMATE), and (3) an automated annotation tool (TYPEMATIC). The compiler introduces tainted types to create a well-typed TYPEFLEXER program.

For sandbox-based SSeFI (S_{all}, Case 2 in Figure 2), CHECKMATE partitions the program into trusted (`c`) and tainted (`u`) source files, compiling the former with the TYPEFLEXER compiler and the latter with the sandbox compiler. The resulting binaries link together to enforce the guarantees in Table II. For other isolation mechanisms, such as I_{mem} and S_{mem}, only the TYPEFLEXER compiler is needed.

A. Compiler and Type Checker

TYPEFLEXER is built atop CLANG, adding 9KLoC of modifications across its parsing (`Parser`), semantic analysis (`Sema`), and instrumentation (`CodeGen`). We incorporate our type well-formedness rules (Section IV) and perform runtime instrumentation (e.g., pointer swizzling, `T_Check`) for I_{mem}, S_{mem}, or S_{all} based on the selected isolation mechanism.

B. Loop Sanity-check Code Motion (LSCM)

Each tainted pointer de-reference includes a check (`T_Check`) to ensure validity, as noted in Section V. Tightly looping pointer accesses can incur high overhead when repeatedly checked. To address this, we introduce *Loop Sanity-check Code Motion* (LSCM), which hoists `T_Check` calls outside bounded-sequential-access (BSA) loops. LSCM employs LICM [31], Lazy Value Information (LVI), and Scalar Evolution (SCEV) analyses to identify BSA loops and hoist all the checks out of such loops. The Listing 4 shows an example of our optimization.

```
void example(_TPtr<reg_t> reg) {
    _TPtr<char> badptr = reg->badptr;
    // after LSCM
    T_Check(badptr, 2378 + 2*(reg->random_bound-1));
    T_Check(badptr, 2*2378 + (reg->random_bound-1));
    for (int i = 0; i < 2379; i++) {
        for (int j = 0; j < reg->random_bound; j++) {
            badptr[i + 2 * j] = badptr[2 * i * j];
            // before LSCM
            T_Check(badptr, i + 2*j);
            T_Check(badptr, 2*i*j);
        }
    }
}
```

Listing 4: Example of LSCM hoisting `T_Check` from loops.

C. CHECKMATE

As mentioned in Section V-C, SANDBOX partitioning mechanism requires the program to be partitioned into `c` region and `u` region source files. CHECKMATE (3K SLoC) helps with this by partitioning the source files of a TYPEFLEXER program.

We provide developers with a set of function qualifier annotations (`_Tainted`, `_Callback`, `_Mirror`, `_TLIB`) to guide the partitioning process (see Appendix B).

CHECKMATE identifies all annotated functions and copies them into a new set of source files, which constitute the `u` region. It then modifies the original code (the `c` region) by inserting special calls called ”sandbox stubs”. These stubs ensure that when the `c` region code calls a function that now resides in the u region, the call is correctly redirected to the u region's library.

However, before the u region source files can be compiled by the sandbox-specific compiler (e.g., WASM), the TYPEFLEXER annotations (such as _TPtr<T> and _Tainted) within these source files must be removed. For this, an auxiliary OCaml tool (680 SLoC), invoked as part of CHECKMATE, is used. Once this tool is run, the generated u region code is in vanilla-C format and can be compiled to a u region library by a sandbox-specific compiler.

D. TYPEMATIC

TYPEMATIC automates pointer annotation by extending 3C [32] to insert tainted pointer qualifiers using flow-sensitive constraints derived from our typing rules (Figure 4).

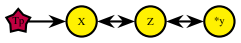
Beginning with developer-supplied tainted seeds, TYPEMATIC propagates taint to related pointers. We implemented **taint3c**, a specialized version of 3C's program analysis technique, to handle this propagation.

Each rewritable pointer instance—e.g., int** has two—is associated with a unique qualifier variable. We traverse the AST and add constraints according to our typing rules (Figure 4).

For instance $x \subseteq y$, where x and y represent qualifier variables or constants (*tainted* or *untainted*). Constraint solving starts by labeling all qualifier variables as *untainted*, then recursively propagates *tainted* to those forced by the constraints. The solution is a final mapping of qualifier variables to their appropriate tainted or untainted state, which we use to rewrite pointers with taint annotations.

For instance:

```
void func(int **y, int* z) {
  _TPtr<int> x;
  z = x;
  *y = z;
}
```



Here, z and $*y$ become tainted, while y remains untainted. The rewritten code is:

```
void func(_Ptr<_TPtr<int>> y,
          _TPtr<int> z) {
  _TPtr<int> x;
  z = x;
  *y = z;
}
```

E. Constraint-Based Taint Explosion Analysis

```
typedef struct basic {
  int* arg1;
  struct basic* arg2;
} t_b;

int main() {
  _TPtr<t_b> basic_t;
  // memory allocation ...
  int* alias1 = basic_t->arg1;
  int* alias2 = alias1;
  int *alias5 = alias2;
  // additional tainted assignments
  alias5 = some_1;
  alias5 = some_2;
  return 0;
}
```

Listing 5: Example illustrating taint explosion.

TYPEMATIC also identifies *taint explosion*, where taint spreads excessively along data-flow paths. Consider the listing here Listing 5. Here, each pointer is assigned an *influence score* quantifying how many pointers it taints.

A base score of 1 is assigned to each leaf node (*e.g.,* some_1 and some_2), which is then propagated upstream (*e.g.,* alias5 will get 2, alias2 gets 3, etc). Pointers with high influence scores (*i.e.,* basic_t) are prime targets for developers to adjust or refactor, thus limiting the spread of taint.

a) Constraint Graph Construction: TYPEMATIC performs a whole-program analysis (taint3c) and constructs a directed taint flow graph $G = (V, E)$ as shown in the listing 6. Here, the nodes V represent pointers or data structures, and the edges E denote the data-flow for taint-propagation between them.

The edge from TPTR to _TPtr<t_b> establishes the initial taint source, which then propagates to struct basic from the type-rules of TYPEFLEXER. From here, taint flows into its fields arg1_0 and arg2_1 and subsequently through all the aliasing pointers.

b) Score Assignment Algorithm: To quantitatively assess each pointer's influence on taint propagation, TYPEMATIC employs a taint scoring algorithm grounded in the constructed graph G. The algorithm initializes leaf nodes—defined as pointers with no outgoing edges and not representing structures—with a base score of 1. It then performs a reverse topological sort to iteratively propagate scores upstream, assigning each non-leaf node a score equal to the sum of its immediate successors' scores. For instance, alias2 aggregates scores from alias3 and alias5, resulting in a cumulative score of 3. Central nodes like struct basic and _TPtr<t_b> achieve higher scores (4 each), underscoring their roles as primary conduits for taint flow within the graph.

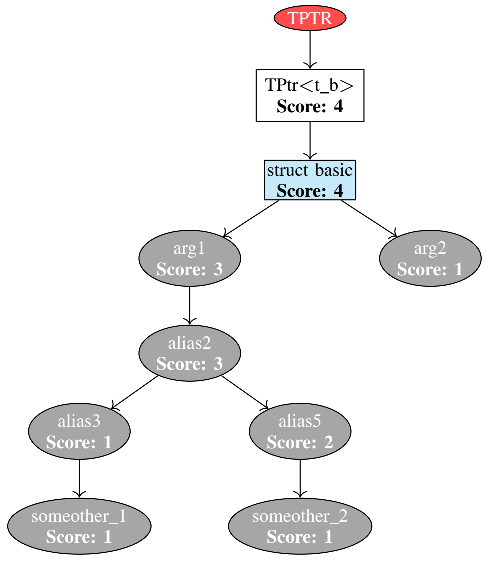


Fig. 6: Taint Propagation Type Reference (TPTR) Graph

c) Analysis Interpretation: In the listing 6, nodes such as struct basic, TPTR, and _TPtr<t_b> attain the highest scores of 4, underscoring their role in orchestrating taint explosion across the graph. By pinpointing struct basic, and `_TPtr<t_b>` as high-scoring nodes, developers can strategically intervene at these junctions (assignments) to disrupt taint propagation. For example, by un-marking `_TPtr<t_b>` as tainted.

In summary, the taint scoring algorithm empowers developers with actionable insights, enabling them to focus their taint mitigation efforts on the most influential pointers within the codebase. This targeted strategy not only enhances the precision of taint analysis but also optimizes resource utilization by avoiding redundant or low-impact annotations.

VII. Evaluation

We evaluate TypeFlexer along four aspects: the effectiveness of TypeMatic in reducing annotation effort (Section VII-A) needed for partitioning, the *runtime and memory overhead* (Section VII-B) (including improvements from LSCM optimizations), a *comparison* with existing systems (Section VII-C), and the *security impact* (Section VII-D) in mitigating real-world vulnerabilities.

Dataset. We use the following three sets of programs to evaluate TypeFlexer.

- *SPEC2006 Benchmarks (DS_1):* We choose CPU-intensive workloads from SPEC2006 (see Table IV) to measure performance and compare with prior work, specifically PtrSplit [33], which also reports results on SPEC2006.
- *Network and Checked C Programs (DS_2):* We primarily select network servers to showcase the benefit of partitioning in security-critical contexts. Additionally, we include Checked C programs from [34] to evaluate memory-partitioning techniques. Table III lists these five programs.
- *Known Vulnerabilities (DS_3):* We gather 16 previously disclosed CVEs (see Table V), chosen based on the following criteria: availability of proof-of-concept (PoC), simplicity of setup, availability of a patch, and availability of benchmarking test suites for validation. Vulnerabilities are labeled S_B (stack buffer overflow), H_B (heap buffer overflow), or U_F (use-after-free).

Experimental Setup. Experiments were conducted on a 6-Core Intel i7-10700H machine with 40 GB RAM, running Ubuntu 20.04.3 LTS. We used WASM [35] as the target sandbox with configurations matching prior work [36].

Performance was evaluated using each program's test suite. For programs without a test suite, their performance is marked as "N/A." Runtime was measured using the elapsed clock cycles via the POSIX `clock()` API and Linux's `time` command, averaging ten runs per measurement. Profiling tools `gprof` and `perf` were used for function-level analysis. Memory usage was measured with Valgrind's "massif" [37], using peak heap usage as the metric.

A. Partitioning Effort with TypeMatic

Partitioning in TypeFlexer involves identifying risky entities—*i.e.,* functions or pointers to be placed in the `u` region—and annotating them accordingly.

Identifying Entities to Partition. We focus on functions that handle user (tainted) data, applying rules derived from prior work [19], [38]:

- *U_LIB:* Unguarded library function calls with user data (*e.g.,* CVE-2014-0160/Heartbleed).
- *U_PA:* Unguarded pointer arithmetic on a user-data pointer (*e.g.,* CVE-2018-19872).
- *U_PC:* Unguarded casting of a user-data pointer (*e.g.,* CVE-2015-7547).
- *U_ULIB:* Unsafe library functions operating on user-data pointers (*e.g.,* CVE-2017-7472).

These rules successfully identify all vulnerabilities in our dataset (DS_3). For network programs (DS_2), they mark most input-processing methods as tainted. In SPEC benchmarks (DS_1), we use the same tainted pointers as PtrSplit.

Annotation Effort. Once developers specify an initial set of tainted pointers, TypeMatic automates the propagation of annotations throughout the code. The TypeMatic *Annotations* column in Table III and Table IV quantify the annotations automatically propagated by TypeMatic from a single function/pointer annotated according to the above rules. Without TypeMatic, developers would have to annotate on an average $\sim$100 pointers per program. Specifically, the Manual Effort column in Table III and IV indicates the amount of time required to manually annotate pointers from an initial set of developer-specified annotations. This process was guided iteratively by TypeFlexer's interactive type-checker, which pinpointed typing violations suggesting the need for annotations. With TypeMatic, these annotations would be automatically added, and the developer just needs to add sanitizations (if needed to avoid taint explosion). These results demonstrate the effectiveness of TypeMatic in reducing the annotation effort.

B. Performance Overhead

We measure overhead for each SSeFI mechanism (see Section V). Table III and Table IV report our results on network programs and SPEC CPU2006 benchmarks, respectively. Memory overhead is specific to each mechanism and is negligible in size.

I_{mem} **Overhead.** I_{mem} (column I_{mem} in Tables III and IV) generally adds minimal overhead, often near 0%, and sometimes even negative. This is because of the Hoard allocator [24] (used to maintain `u` region), which is more efficient than the traditional allocator (*i.e.,* `malloc`) for certain use cases, resulting in more efficient execution than the original program. Most overhead originates from `T_Check`, which validates each tainted pointer's membership in the `u` region heap. Although invoked on every tainted pointer access, its impact depends more on access patterns and LSCM optimizations than on pointer count (*e.g.,* `lbm` with 681 pointers vs. `libquantum` with 182). The **W/O LSCM** and **W LSCM** columns in Tables III and IV show the performance overhead without and with Loop Sanity-check Code Motion (LSCM), respectively. We can see that LSCM significantly reduces the overhead for most benchmarks, with the majority of them to 0%. For instance, for `libquantum` (in Table IV), LSCM reduced the overhead from 217% to 35%, a 3X reduction.

Sandbox Dependent Isolation Mechanisms. The overhead of other SSeFI mechanisms (S_{mem}, S_{all}) mainly depend on the target sandboxing mechanism. As mentioned before, we used WASM, which, as shown by the recent work [39], can introduce up to 200% overhead. We also performed micro benchmarking and validated this. More details in (Appendix C).

S_{mem} **Overhead.** The S_{mem} column of Tables III and IV, show the overhead of S_{mem} mechanism. The glaring observation in Table III (e.g `libPNG`) sustains a negative overhead. This is because the memory allocated by these custom allocators (S_{mem} and I_{mem}) yields a faster overall access speed. As expected, for most of the cases, the overhead is slightly greater than I_{mem} because, as explained in Section V-B: (i) Tainted allocations must switch to the sandbox and (ii) Tainted pointers require swizzling. However, in a few cases (*e.g.,* `libquantum` in Table IV and `Parsons` in Table III), the overhead is lower than I_{mem}. This is because, unlike in I_{mem}, where `T_Check` check involves checking tainted memory ranges, `T_Check` in S_{mem} is just a single range (*i.e.,* sandbox memory range) check. Consequently, for applications with more `T_Check` calls, the additional cost in I_{mem} surpasses the overhead related to sandbox switching, resulting in high overhead. We provide a more detailed comparison in (Appendix C1).

S_{all} **Overhead.** S_{all} (Section V-C) isolates both code and data of the `u` region within the WASM sandbox. Consequently, Table IV and Table III show it incurs higher overhead than I_{mem} or S_{mem}. Programs like `lbm`, which spend most of their runtime in the sandbox, incur the largest overhead. As mentioned before, the overhead stems primarily from WASM sandbox execution rather than additional checks in TYPEFLEXER.

Taint Explosion Analysis. We manually analyzed the results of our taint explosion analysis (Section VI-E) and found that all the identified explosion points are correct. For instance, in SPEC CPU2006, taint explosion occurs mainly in `lbm` and `libquantum`. TYPEMATIC correctly identified `srcGrid`/`dstGrid` (`lbm`) and `quantum_reg`/`quantum_reg_node` (`libquantum`) as core tainted objects. The CDFs (Cumulative Distribution Function) generated by TYPEMATIC on SPEC2006 benchmarks reveal critical insights into TYPEMATIC's dynamics. Figure 7, shows the CDFs of tainted pointers for `libquantum` and sphinx3.

`libquantum` exhibits a steep increase in the CDF, which indicates taint explosion. This is due to the pervasive influence of the `Tstruct quantum_reg`, which makes all member pointers and external pointers to this structure tainted.

In contrast, `sphinx3` follows a linear trend. This linearity signifies that the initial taint annotations have a direct and contained influence. This linearity reflects an absence of secondary taint propagation, highlighting that taint effects are confined to their immediate data interactions.

The graph clearly demonstrates the impact of initial annotations on taint propagation. Specifically, annotations targeting pointers to complex structures or entire structures significantly amplify taint spread, highlighting their profound influence on overall taint propagation.

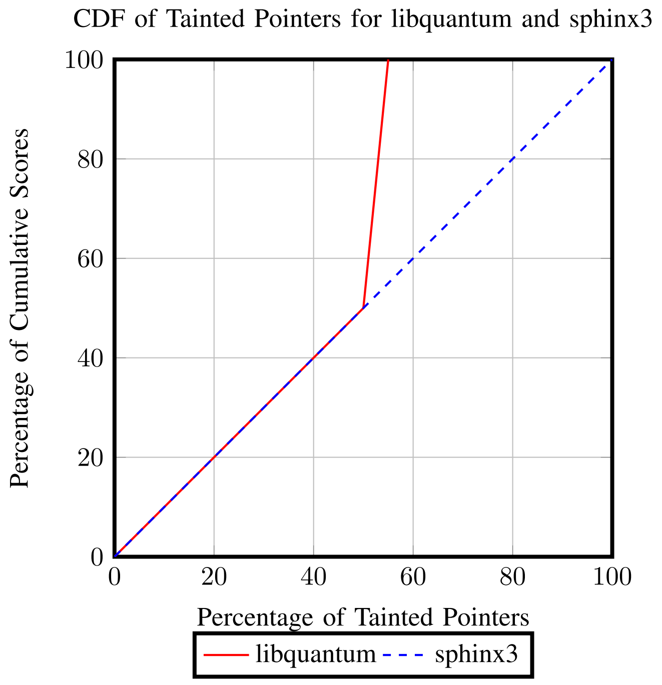


Fig. 7: Combined CDF of Tainted Pointers for libquantum and sphinx3 with distinct line styles

C. Comparison with Related Work

We selected PTRSPLIT [33] for our comparison as it supports general pointers and doesn't require special hardware features, such as SGX (for GLAMDRING [8]). As mentioned before, the source code of PTRSPLIT is unavailable, and only performance numbers with SPEC2006 were available.

The last column of Table IV shows the runtime overhead of PTRSPLIT on various programs of SPEC2006. *Except for* `lbm`, *the overhead of all our SSeFI isolation mechanisms is lower than that of* PTRSPLIT. *This is even true for the case of* S_{all}, *which provides the same guarantees of* PTRSPLIT.

For `lbm`, the high overhead of 844.1% arises because a significant portion of the program is marked as tainted and thus fully executes within the WASM sandbox, which introduces a runtime overhead of 1434% (our micro-benchmarks (Appendix C)).

Using perf, we observed that approximately 95% of the execution time occurs within the WebAssembly sandbox, making it the primary contributor to the overall overhead. We emphasize that these performance limitations are attributable to the sandbox itself, not to TYPEFLEXER. Therefore, selecting more efficient sandboxes can enhance performance.

D. Security Impact

Previously Known Vulnerabilities (DS_3). We first applied the guidelines from Section VII-A to mark initial tainted functions, then used TYPEMATIC to propagate taint annotations.

TABLE III: **Results of TypeFlexer on Network-program dataset (DS_2)**

Program	Description	Size (SLoc)	Manual Effort (hrs)	TypeMatic Annotations		SeFI Runtime Overhead						CVEs Isolated
				`u` region		I_{mem}		S_{mem}		S_{all}		
				Num pointers	LOC	W/O LSCM	W LSCM	W/O LSCM	W LSCM	W/O LSCM	W LSCM	
ProFTPD	High-performance FTP server	556K	1	6	0	0.31%	0.29%	0.41%	0.38%	N/A	N/A	CVE-2010-4221 (✓)
MicroHTTPD	Simple HTTP server	122K	3	139	450	0.4%	0%	0%	0%	0%	0%	N/A
UFTPD	UDP-based FTP server	3K	3	146	90	0%	0%	0%	0%	2.3%	2.3%	CVE-2020-14149 (✓) CVE-2020-5204 (✓)
LibPNG	png2pnm & pnm2png	76K	8	248	0	-3.2%	-3.8%	-7.65%	-8.33%	N/A	N/A	CVE-2018-144550 (✓)
Parsons	JSON-parsing library	3.1K	5	364	800	13.48%	12.3%	-3.98%	-9.2%	262%	262%	N/A
Geometric Mean/**Median**		34.36K/**76K**	3.24/**3**	102/**146**	318/**90**	2.19%/**0.31%**	1.76%/**0%**	-2.2%/**0%**	-3.43%/**0%**	88.1%/**2.3%**	88.1%/**2.3%**	

TABLE IV: **Results of TypeFlexer on SPEC2006 Benchmarks (DS_1). W/O LSCM and LSCM indicate performance without and with LSCM optimization, respectively.**

Program	Size (SLoc)	Sensitive Pointers (Initial Annotations)	Manual Effort (hrs)	TypeFlexer							PtrSplit
				TypeMatic Annotations		SeFI Runtime Overhead					
				Num pointers (Total)	Num of functions (Total)	Imem		Smem		Sall	
						W/O LSCM	W LSCM	W/O LSCM	W LSCM	W LSCM	
lbm	1.1 K	LBM_GridPtr LBM_Grid	6	681 (695)	5 (19)	13.7%	0%	8.4%	0%	844.1%	24.3%
libquantum	4.3 K	All quantum_reg* All quantum_reg_node* hash	6	182 (1,690)	6 (115)	217.5%	35.4%	35%	0%	39.2%	179.2%
h264ref	51.5 K	FirstMBInSlice	1	10 (32,212)	5 (590)	0.38%	0%	1.18%	0%	2.36%	15.5%
bzip2	8.3 K	progName	0.3	8 (4,356)	6 (100)	3.7%	0%	4%	0%	N/A	5.3%
sjeng	13.5 K	realholdings	1	2 (3,415)	5 (144)	1%	0%	1%	0%	N/A	10.2%
milc	13.5 K	path_coeff act_path_coeff	0.3	7 (5,001)	2 (235)	5.8%	0%	6.9%	0%	N/A	2.2%
sphinx3	25K	liveargs	0.2	9 (949)	3 (369)	1.1%	0%	0.8%	0%	N/A	7.1%
Geometric mean			**0.94**	(899)**48,318**	(32)**1,572**	**4.28%**	**0%**	**3.58%**	**0%**	**43.1%**	**12.8%**

Abbreviations: "Num of functions": number of functions whose program dependence graph contains the sensitive pointer. "LSCM": Loop Sanity-check Code Motion.

All vulnerable functions were ultimately classified as tainted. We chose I_{mem} and S_{mem} as isolation mechanisms for each partitioned program. Next, we confirmed each vulnerability and its affected pointers resided in the tainted region, and compiled the partitioned programs under I_{mem} or S_{mem}. We then tested them using publicly available exploits and bug-triggering inputs. As summarized in Table V (column ✓), all exploits failed to compromise the host `c` region, demonstrating effective isolation.

Network and Checked C Programs (DS_2). We also reintroduced older vulnerabilities in three network-related or Checked C-based programs. Again, we confirmed the exploit code could not escape isolation, as indicated by ✓ in Table III. These findings confirm that TypeFlexer's SSeFI mechanisms effectively mitigate real-world vulnerabilities. We discuss CVE-2017-9204 as a case study (Appendix D).

When taint explosion became an issue, we used marshalling to isolate vulnerable pointers, avoiding widespread pointer annotations.

VIII. Related Work

Program Partitioning Mechanisms. Several works partition applications into trusted and untrusted components [5], [6], [7], [8], [9], [13], often following a *data-centric* paradigm [8], [9]. These methods usually involve marshaling (serialization/deserialization) and incur 37%–163% overhead [8], [9]. Many solutions remain bound to a single isolation mechanism, limiting deployment flexibility. RLBox [36] also employs tainted types but focuses on C++ templates and library-level isolation, complicating its application to arbitrary C code. RLBox lacks automated type annotation or formal isolation guarantees and only supports complete library partitioning.

Retrofitting New Type Systems. Existing safe-C dialects [23], [40], [41], [42], [43], [44] enhance spatial or temporal safety, often through "fat" pointers or shadow metadata [40], [41], thereby adding runtime overhead. Cyclone [42] and Deputy [44] similarly refine memory safety but do not provide sandbox-based isolation. In contrast, TypeFlexer's SSeFI mechanisms use taint-based *partitioning* without "fat" pointers, letting developers choose among multiple sandbox backends based on desired overhead and isolation. This decoupled architecture allows flexible trade-offs, ensuring comprehensive type-driven

TABLE V: **Results of TYPEFLEXER on Vulnerability dataset (DS_3).** We use S_B (Stack buffer overflow), H_B (Heap buffer overflow), and U_F (Use after free) to indicate corresponding vulnerability types.

CVE-ID (Type)	**Program**	**Size (SLoc)**	**Functions (Pointers)**	I_{mem}	S_{mem}
CVE-2016-10094 (S_B)	LibTIFF-v4.0.7	185 K	1 (6)	✓	✓
CVE-2017-9204 (S_B)	imageworsener-1.3.1	70 K	1 (8)	✓	✓
CVE-2017-9205 (S_B)	imageworsener-1.3.1	70 K	1 (8)	✓	✓
CVE-2015-8668 (S_B)	tiff-4.0.1	1,845 K	1 (3)	✓	✓
CVE-2004-1257 (S_B)	abc2mtex1.6.1	5K	1 (1)	✓	✓
CVE-2010-2891 (H_B)	LibSMI	187 K	5 (19)	✓	✓
CVE-2012-4409 (H_B)	mcrypt-2.5.8	40 K	1 (4)	✓	✓
CVE-2004-1120 (H_B)	prozilla	24 K	1 (2)	✓	✓
CVE-2004-1292 (H_B)	ringtonetools	7 K	1 (2)	✓	✓
CVE-2004-2093 (H_B)	rsync	34 K	1 (4)	✓	✓
EDB-14904 (H_B)	fcrackzip	12 K	1 (4)	✓	✓
CVE-2006-0539 (H_B)	fcron	33 K	1 (2)	✓	✓
CVE-2010-2089 (U_F)	Python-2.6	917 K	1 (2)	✓	✓
CVE-2014-4616 (U_F)	Python-2.7.1	1,014 K	2 (5)	✓	✓
CVE-2015-7805 (U_F/H_B)	libsndfile-1.0.25	112 K	1 (2)	✓	✓
CVE-2010-1634 (U_F)	Python-2.7	1013 K	1 (2)	✓	✓

partitioning while remaining agnostic to the specific sandbox technology.

IX. LIMITATIONS AND FUTURE WORK

We acknowledge the following limitations and plan to tackle them as part of future work.

- **Sandbox Dependency:** TYPEFLEXER assumes the availability of a sandbox and consequently inherits all the limitations of the corresponding sandbox. *e.g.,* Programs should be compilable with the sandbox compiler. Also, as shown in Section VII-B, the performance of the partitioned applications mainly depends on the sandbox. However, our implementation is not dependent on one specific sandbox and can be easily extended to other sandboxes. As future work, we will extend our implementation to other sandboxes.
- **Taint Explosion:** Although, TYPEMATIC helps in identifying taint explosion points, it does not provide ways to handle it. Developers need to figure out appropriate sanitization routines and place them at these explosion points, which might be inconvenient. We plan to explore automated sanitization routines insertion techniques as part of our future work.
- **Intra-`u` region Corruption:** TYPEFLEXER guarantees that tainted pointers cannot corrupt the `c` region without detection, but offers no protection against attacks that stay entirely within the `u` region.
- **Initial Annotations:** Security of a program using TYPEFLEXER hinges on complete and accurate taint seeds provided by the developer. Omitted seeds (*e.g.,* developer oversight) may leave vulnerable/risky pointers unchecked, resulting in undertainting.
- **CLANG and 3C Limitations:** Our implementations of type checker and TYPEMATIC are based on CLANG's type checker and 3C, respectively. Consequently, we inherit the corresponding frameworks' and tools' limitations. Although we have extensive tests (300), we cannot guarantee bug-free implementation because of the fundamental limitation of testing.
- **Benchmark Representativeness:** Our evaluation focuses on user-space programs. As part of our future work, we plan to extend our system to other domains, such as Embedded firmware, kernel modules, and hand-tuned code with custom allocators.

X. CONCLUSION

We present TYPEFLEXER, a flexible type-directed program partitioning system that effectively separates policy and mechanism by introducing a type-system that enables fine-grained program partitioning. We formalize the type-system and semantics and prove various safety properties ensuring spatial memory safety through isolation, Its implementation enables developers to partition applications interactively and effectively. Our evaluation over multiple datasets shows that TYPEFLEXER provides an effective and efficient technique for program partitioning with almost no overhead for most isolation mechanisms compared to 12.8% by the related work [33]. We also show TYPEFLEXER is successful in isolating several security vulnerabilities.

ACKNOWLEDGMENTS

We want to thank our shepherd and anonymous reviewers for their valuable comments and inputs to improve our paper. This research was supported in part by the National Science Foundation (NSF) under Grants CNS-2340548, CPS-2333487, and CCF-2422127. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] C. Trends, "Cve trends," https://www.cvedetails.com/vulnerabilities-by-types.php, 2021, accessed: 2020-10-11.
[2] BlueHat, "Memory corruption is still the most prevalent security vulnerability," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019, accessed: 2020-02-11.
[3] B. Zeng, G. Tan, and U. Erlingsson, "Strato: A Retargetable Framework for Low-level Inlined-reference Monitors," in *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[4] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, “SoK: Sanitizing for security,” in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[5] G. Tan *et al.*, “Principles and implementation techniques of software-based fault isolation,” *Foundations and Trends® in Privacy and Security*, vol. 1, no. 3, pp. 137–198, 2017.

[6] D. Brumley and D. Song, “Privtrans: Automatically partitioning programs for privilege separation,” in *USENIX Security Symposium*, vol. 57, no. 72, 2004.

[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp, “Wedge: Splitting applications into reduced-privilege compartments.” USENIX Association, 2008.

[8] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 285–298.

[9] 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.

[10] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[11] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, “Gobi: Webassembly as a practical path to library sandboxing,” *arXiv preprint arXiv:1912.02285*, 2019.

[12] M. Payer, “Software security: Principles, policies, and protection,” 2019.

[13] S. Rul, H. Vandierendonck, and K. De Bosschere, “Towards automatic program partitioning,” in *Proceedings of the 6th ACM conference on Computing frontiers*, 2009, pp. 89–98.

[14] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, “Dynpta: Combining static and dynamic analysis for practical selective data protection,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1919–1937.

[15] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, “Preventing kernel hacks with hakcs.” in *NDSS*, 2022, pp. 1–17.

[16] C. C. Spec, “The Checked C,” https://github.com/microsoft/checkedc, 2016, accessed: 2020-10-10.

[17] 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.

[18] D. Meng, M. Guerriero, A. Machiry, H. Aghakhani, P. Bose, A. Continella, C. Kruegel, and G. Vigna, “Bran: Reduce vulnerability search space in large open source repositories by learning bug symptoms,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 731–743.

[19] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, “Leopard: Identifying vulnerable code for vulnerability assessment through program metrics,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 60–71.

[20] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, “Velvet: a novel ensemble learning approach to automatically locate vulnerable statements,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 959–970.

[21] L.-Y. Situ, Z.-Q. Zuo, L. Guan, L.-Z. Wang, X.-D. Li, J. Shi, and P. Liu, “Vulnerable region-aware greybox fuzzing,” *Journal of Computer Science and Technology*, vol. 36, pp. 1212–1228, 2021.

[22] S. Blazy and X. Leroy, “Mechanized Semantics for the Clight Subset of the C Language,” *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10817-009-9148-3

[23] L. Li, Y. Liu, D. L. Postol, L. Lampropoulos, D. V. Horn, and M. Hicks, “A formal model of Checked C,” in *Proceedings of the Computer Security Foundations Symposium (CSF)*, Aug. 2022.

[24] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications,” *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[25] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Heaphopper: Bringing bounded model checking to heap implementation security,” in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 99–116.

[26] G. Blankenagel and R. H. Güting, “External segment trees,” *Algorithmica*, vol. 12, no. 6, pp. 498–532, 1994.

[27] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[28] A. Milanova, A. Rountev, and B. G. Ryder, “Precise call graph construction in the presence of function pointers,” in *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2002, pp. 155–162.

[29] D. Ahmad and I. Arce, “The confused deputy and the domain hijacker,” *IEEE Security & Privacy*, vol. 6, no. 1, pp. 74–77, 2008.

[30] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, “Boomerang: Exploiting the semantic gap in trusted execution environments.” in *NDSS*, 2017.

[31] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers principles, techniques, and tools addison-wesley, 1986,” *QA76*, vol. 76, no. C65A37, p. 1985, 1985.

[32] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, “C to checked c by 3c,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–29, 2022.

[33] S. Liu, G. Tan, and T. Jaeger, “PtrSplit: Supporting general pointers in automatic program partitioning,” in *CCS*. ACM, 2017.

[34] Microsoft, “Benchmarks for evaluating Checked C,” https://github.com/microsoft/checkedc/wiki/Benchmarks-for-evaluating-Checked-C, 2019, accessed: 2020-10-27.

[35] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” in *PLDI*. ACM, 2017.

[36] 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)*. USENIX Association, Aug. 2020, pp. 699–716. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[37] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3-advanced debugging and profiling for gnu/linux applications*. Network Theory Ltd., 2008.

[38] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, “Understanding the reproducibility of crowd-reported security vulnerabilities,” in *27th {USENIX} security symposium ({USENIX} security 18)*, 2018, pp. 919–936.

[39] A. Jangda, B. Powers, E. D. Berger, and A. Guha, “Not so fast: Analyzing the performance of {WebAssembly} vs. native code,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 107–120.

[40] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “CCured: Type-Safe Retrofitting of Legacy Software,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, 2005.

[41] 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*, 2009.

[42] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, , and Y. Wang, “Cyclone: A Safe Dialect of C,” in *USENIX Annual Technical Conference*. Monterey, CA: USENIX, 2002, pp. 275–288.

[43] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, “Region-based Memory Management in Cyclone,” in *PLDI*, 2002.

[44] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “SafeDrive: Safe and recoverable extensions using language-based techniques,” in *7th Symposium on Operating System Design and Implementation (OSDI’06)*. Seattle, Washington: USENIX Association, 2006.

[45] B. J. Nelson, *Remote procedure call*. Carnegie Mellon University, 1981.

[46] V. Costan and S. Devadas, “Intel sgx explained,” *Cryptology ePrint Archive*, 2016.

[47] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, “Checked C: Making C Safe by Extension,” in *2018 IEEE Cybersecurity Development (SecDev)*, 2018, pp. 53–60.

[48] G. J. Duck and R. H. Yap, “Heap bounds protection with low fat pointers,” in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 132–142.

[49] J. Duan, Y. Yang, J. Zhou, and J. Criswell, “Refactoring the FreeBSD kernel with Checked C,” in *Proceedings of the 2020 IEEE Cybersecurity Development Conference (SecDev)*, 2020.

[50] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," 2022.

APPENDIX

A. Program Partitioning Background

For instance, in the following code `key` is marked as sensitive and functions that access it, *i.e.,* `initkey` and `encrypt` are considered to be part of trusted partition and the other function, *i.e.,* `main` will be in the untrusted partition.

```
1  char __attribute__((annotate("sensitive"))) *key;
2
3  void initkey(int sz) {
4   ...
5   for (i=0; i<sz; i++) key[i]= ...;
6  }
7
8  void encrypt(char *ptxt, int sz) {
9   ..
10  for (i=0; i<sz; i++)
11   ctxt[i]=ptxt[i] ^ key[i];
12  ..
13 }
14
15 int main() {
16  ..
17  initkey(strlen(txt));
18  encrypt(txt, strlen(txt));
19  ...
20 }
```

As we can see, partitioning is achieved easily in the existing techniques by identifying which functions access sensitive data. These partitions can be separated in various ways, *e.g.,* as different processes communicating through RPC [45], parent and child processes communicating through shared memory, etc.

The main novelty of existing techniques is using various program analysis and dynamic instrumentation methods to facilitate transparent communication between functions in trusted and untrusted partitions. Because of their focus on inter-partition interactions, the existing techniques are specialized for the target isolation mechanism. For instance, PTRSPLIT [9] technique assumes that partitions are hosted as separate processes and data is exchanged through marshaling, *e.g.,* sending data pointed by `ptxt` and `sz` during calls to `encrypt` (*i.e.,* line 18 in the above example) and synchronizing the data back to `ctxt` on return. Any other isolation mechanism or data exchange (*e.g.,* shared memory) cannot be supported as the partitioning technique is specialized for marshaling by using PDG and tracking bounds information. Similarly, GLAMDRING [8] is specialized for partitioning using SGX-enclaves [46].

B. Additional Function Qualifiers

In addition to the `_Tainted` qualifier that marks functions to be in `u` region, we provide a few other qualifiers that enable developers to provide additional information and ease the partitioning process. Specifically, we provide three additional qualifiers: `_Callback`, `_Mirror`, and `_TLIB`.

_Callback: Developers should use this qualifier to mark callback functions, *i.e.,* functions in `c` region, that can be called from the tainted region. This is also used to enforce CFI as explained in Section V-A1. For S_{all} and C_{box} mechanisms, the CHECKMATE inserts appropriate sandbox dependent mechanisms to enable this.

_Mirror: This qualifier permits copying the corresponding function into both `c` region and `u` region, which permits the handling of certain simple utility functions that are called from both regions. An example of such usage is shown in Appendix E.

_TLIB: This qualifier relaxes type-checking rules on library functions, allowing developers to use the functions freely with tainted types. Specifically, the annotated library functions can be called with tainted types. We assume that developers would add checks to ensure that calling the annotated library functions is fine. We provide an example of the usage

C. WASM *Micro-Benchmarking*

Figure 8 shows our micro-benchmarking result. We measure the following operations as part of this:

Memory access in WASM Sandbox (SBX_m): All memory accesses in a sandbox need additional verification by the sandbox runtime, which results in runtime overhead. We perform 100K memory accesses (read and write) in a loop, measure the time inside the sandbox, and compare it with the time executed as a regular program. The results (Figure 8) show that we incur 1434% overhead for memory accesses in the WASM sandbox compared to that of native C code. Although recent works [39] suggest a lower runtime overhead we follow a different approach by first compiling our tainted source code to a wasm binary. We then use wasm2c to generate WebAssembly C code which we later compile alongside wasm-runtime to form our final tainted library. This approach grants us ease in interoperability with C programs as the tainted functions can now be treated as statically linked functions. However, it's important to note that TYPEFLEXER does not restrict users from using wasm-runtime linking.

Sandbox Roundtrip (SBX_{RT}): We measure the time to make a round trip between `c` region and sandbox (`u` region) compared to a regular function call and return. We create a no-op function below:

```
void noop() { return; }
```

We place this `noop` function in the sandbox and measure the time to call and return from it:

```
s = clock(); sandbox_noop(); e = clock();
```

We compare the time with a regular call when `noop` is in `c` region.

As shown in Figure 8, we incur an overhead of $\sim 53\%$. This is much smaller than the overhead reported by prior works [39], [36] as we use binaryen's wasm optimizer to optimize the compiled wasm binary. The reported overhead is because of transitions to/from sandbox require context switches which are more expensive than regular function calls (*i.e.,* `call` and `ret` instructions).

Tainted Pointer Access in `c` region (TP_c): As explained in Section V-B, we need to perform pointer swizzling to convert the sandbox-specific representation of tainted pointers to raw addresses. In addition, our instrumentation also checks at runtime that all tainted pointers are within the sandbox address range before accessing them. We measure this additional

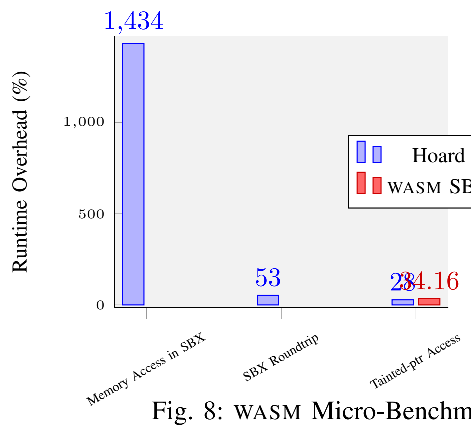


Fig. 8: WASM Micro-Benchmarks.

overhead by comparing tainted pointer accesses with regular pointer accesses.

1) I_{mem} v/s S_{mem} tainted pointer access: As shown in Figure 8, incurred overhead in accessing tainted pointers in `c` region is lower for I_{mem} (28%) as compared to 34% for S_{mem} due to the additional opaque pointer translation cost (eq: 2).

Overhead comparison amongst different partitioning mechanisms: We first create a `perform_loop`:

```
void perform_loop(_TPtr<int> integer_array, int sz) {
    int interval = sz / 10;
    for (int i = 0; i < 10000; i++) {
        for (int j = 0; j < 10; j++) {
            int index = j * interval;
            integer_array[index] += (j % 2 == 0) ? 1 : -1;
        }
    }
}
```

We then call `perform_loop` in a loop of increasing memory allocation sizes and record time as follows:

```
for (int size = start_size;
size <= end_size; size += step_size) {
     s = clock();
     _TPtr<int> array = __malloc__(size * sizeof(int));
     perform_loop(array, size);
     __free__(array);
     e = clock();
 }
```

Although I_{mem}'s per pointer transaction cost is lesser than S_{mem} (fig fig. 8), I_{mem} has higher overall overhead as shown in Figure fig. 9. This is because, I_{mem}'s suffers from frequent cache-miss costs when dereferencing pointers with bad spatial locality. Consequently, programs that have large `u`-region with non-linear tainted pointer accesses benefit more by using S_{mem} instead of I_{mem}.

D. CVE Examples

Listing 1: CVE-2014-0160 (Heartbleed)

```
typedef struct {
    uint16_t type;
    uint16_t payload;
    uint8_t *data;
} tls1_heartbeat;

int tls1_process_heartbeat(SSL3_RECORD *r) {
    unsigned char *p = &r→data[0], *end =
        &r→data[r→length];
    uint16_t hbtype, payload;
    tls1_heartbeat hb;

    hbtype = (p[0] << 8) | p[1];
    payload = (p[2] << 8) | p[3];
    hb.type = hbtype;
    hb.payload = payload;
    p += 4;

    if (hb.type == 1) {
        if (p + hb.payload + 2 > end) {
            return -1;
        }

        hb.data = malloc(hb.payload + 3);
        memcpy(hb.data, p, hb.payload);
    }
    ...
}
```

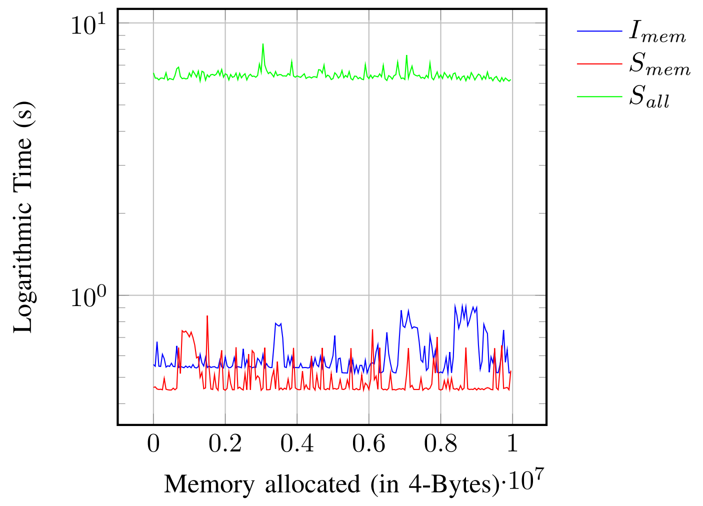


Fig. 9: Partitioning mechanisms and their overhead

Listing 2: CVE-2018-19872 (libGD)

```
typedef struct {
    int width, height;
    unsigned char *pixels;
} gdImage;

int gdImageCreateFromGifCtx(gdImage *image, FILE
     *infile) {
    unsigned char buf[16];
    unsigned char color_table[256 * 3];
    int color_table_size;

    // ... Reading the GIF header ...

    // Reading the color table size from the header
    fread(buf, 1, 3, infile);
    color_table_size = 2 << (buf[2] & 0x07);

    // Reading the color table
    fread(color_table, 1, color_table_size * 3, infile);

    // ... Processing the GIF image ...
}
```

Listing 3: CVE-2017-7472 (Mercurial)

```
#define MAX_CONFIG_LEN 1024

void hg_extension(const char *config) {
    char buf[MAX_CONFIG_LEN + 1];

    // ... Processing other command-line options ...

    // Copying the configuration string to the buffer
    strcpy(buf, config);

    // ... Processing the configuration ...
}
```

Listing 4: CVE-2015-7547(glibc)

```

void process_answer(unsigned char *answer, int
     answer_length) {
    int offset = 0;
    while (offset < answer_length) {
        ....

        rdlength = ntohs(rdlength);
        offset += 10;

        if (rrtype == ns_t_a || rrtype == ns_t_aaaa) {
            struct sockaddr_storage addr;
            if (rrtype == ns_t_a) {
                struct sockaddr_in *addr4 = (struct
                     sockaddr_in *)&addr;
                memcpy(&addr4->sin_addr, answer + offset,
                     sizeof(addr4->sin_addr));

            } else {
                struct sockaddr_in6 *addr6 = (struct
                     sockaddr_in6 *)&addr;
                memcpy(&addr6->sin6_addr, answer +
                     offset, sizeof(addr6->sin6_addr));

            }
        }
        offset += rdlength;
    }
}
```

E. Additional Qualifiers Examples

_Mirror: This qualifier permits copying the corresponding function into both c region and u region, which permits the handling of certain simple utility functions that are called from both regions. For example, append_string in our evaluation of parsons_wasm has callers from both the regions.

```
_Mirror int append_string(_TPtr<char> buf,
const char* appendStr : itype(_Nt_array_ptr<const char>),
_TPtr<char> buf_start, size_t buf_len) {
/* Qualifier Rules:
1.) No access to global data NOT marked "const"
2.) Callees must be _Tainted or _Mirror
*/
...
}
```

Qualifying append_string with _Mirror duplicates the function in both regions, allowing calls to append_string with parameter to appendStr as an unchecked or checked pointer within u and c regions, respectively. Consequently complexity from over-tainting is avoided as appendStr need not be tainted in c region and neither are callbacks required to access append_string from u region. "_Mirror" enforces control-flow and data-flow compile-time semantic rules to ensure all variable and function call dependencies of mirrored functions required for u region's compilation are resolved.

_TLIB: This qualifier relaxes type-checking rules on library functions, allowing developers to use the function freely in c region.

```
// First, manually check the memory is in tainted region.
// if yes, then call strncpy.
if (!is_mem_in_range(t_str, t_str + n, SBX_LOW(), SBX_HIGH()))
  handle_violation();
// our type checker ignores this because
// the _TLIB annotation below.
strncat(dst, t_str, n);
- extern char *strncat (char *__restrict __dest,
+ _TLIB extern char *strncat (char *__restrict __dest,
const char *__restrict __src, size_t __n); // In the header file
}
```

Passing tainted pointer t_str to unqualified strncat above is disallowed without having additional u region implementation for strncat. If a user ascertains that t_str has the right buffer size for strncat, she might label strncat with _TLIB, so that t_str can be treated as an checked pointer parameter; such annotation relaxes type-checking for all the arguments to its calls. It is worth noting that TYPEFLEXER does not enforce any semantics to ensure _TLIB functions implemented in c region are non memory-modifying; therefore, using _TLIB requires users' awareness of memory address leaks.

F. Generating c region Source Partition

Handling Calls to Tainted Functions: In c region, we also need to modify calls to tainted functions as they execute inside the sandbox (separate address space) and thus cannot be invoked as regular functions. However, modifying every call site of tainted functions is tedious and also requires precise pointer analysis [28] to handle indirect calls through function pointers.

We handle this by *indirection*: Instead of modifying the call sites, we modify the body of tainted functions to invoke the corresponding function in the sandbox. For instance, we modify the body of tainted function process_req1(from Figure 1) in c region as below:

```
_Tainted int process_req1(_TPtr<char> msg, size_t m_l){
- int rc = -1, i;
- if (m_l > MIN_SIZE) {
- ...
+ return w2c_process_req1(msg, m_l);
}
```

This ensures that all calls (even through function pointers) to the tainted function process_req1 are redirected to the sandbox.

Handling _Callback Qualifiers: Consider the following StringAuth function that checks whether the provided user input usertoken is authenticated by accessing checked data. Since this needs to be invoked from u region it is annotated as a _Callback.

```
_Callback _TPtr<char> StringAuth(
                _T_Array_Ptr<const char> usertoken : count(len),
                size_t len) {
```

```
...
// Checks whether usertoken is authenticated
/*
 These functions will be restricted to only accept
 tainted parameters.
*/
...
}
```

These callback functions are only allowed to use tainted parameters as they will be called from a tainted region.

For each such function, we create a corresponding trampoline function that serves as the entry point for the callback function, as shown below:

```
+ unsigned int _T_StringAuth(void* sandbox,
+               unsigned int arg_1,
+               unsigned long int arg_2) {
+   // Perform necessary Type-conversion of arguments.
+   // uname <- conver arg_1
+   // len <- arg_2
+   ret = StringAuth(uname, len);
+   // ret_val <- ret
+   return ret_val;
+ }
```

The trampoline function handles the invocations from sandbox (and hence the extra parameter `sandbox`), performs necessary pointer argument conversion, and eventually invokes the callback.

We also add the code to register this trampoline function with the sandbox. The registration function for WASM sandbox is as shown below:

```
+ void registerCallback_StringAuth(void){
+ //callback function signature {ret <- int, arg_1 <- int, arg_2 <- long}
+ int ret_param_types[] = {0, 0, 1};
+ // 2 <- arg count, 1 <- ret count
+ __StringAuth__C_ = _SBXREG_((void*)_T_StringAuth,2,1, ret_param_types);
+ }
```

This registration function creates an opaque handle for the trampoline function and enables `u` region to call the trampoline using the corresponding handle.

Lastly, we change the tainted function's body to include an indirect call to the sandbox's implementation of the tainted function. However, instead of passing the callback function pointer directly from the argument list, we pass the generated trampoline handle `__StringAuth__C_` as shown below:

```
_Tainted _TPtr<char> StringProc(_TPtr<_TPtr<const char>> user_input,
_TPtr<_TPtr<char>(_TPtr<const char> input, size_t len)>StringAuth) {
-  ...
-  //complex Function Body
- return StringAuth(one_past_start, string_len);
+ return w2c_StringProc(_SBX_(), (unsigned int)string, __StringAuth__C_);

}
```

G. *Checked C*

Recently, Elliott *et al.* [47] and Li *et al.* [23] introduced and formalized Checked C, an open-source extension to C, to ensure a program's spatial safety by introducing new pointer types, *i.e.,* checked pointer types. Which are represented as system-level memory words without "fattening" metadata [48], and ensuring backward compatibility, *i.e.,* developers can use checked and regular (unchecked or wild) pointers within the same program.

Checked Pointer Types. Checked C introduces three varieties of *checked pointer*:

- `_Ptr`<T> (ptr) types a pointer that is either null or points to a single object of type T.
- `_Array_ptr`<T> (arr) types a pointer that is either null or points to an array of T objects. The array width is defined by a *bounds* expression, discussed below.
- `_NT_Array_ptr`<T> ($ntarr$) is like `_Array_ptr`<T> except that the bounds expression defines the *minimum* array width—additional objects may be available past the upper bound, up to a null terminator.

Both arr and $ntarr$ pointers have an associated bounds which defines the range of memory referenced by the pointer. The three different ways to declare bounds and the corresponding memory range is:

`_Array_ptr`<|T|> `p: count`(n—)—	$[\mathtt{p}, \mathtt{p} + \mathtt{sizeof}(T) \times n)$
`_Array_ptr`<|T|> `p: byte_count`(b—)—	$[\mathtt{p}, \mathtt{p} + b)$
`_Array_ptr`<|T|> `p: bounds`(x, y—)—	$[x, y)$

The bounds can be declared for $ntarr$ as well, but the memory range can extend further to the right, until a `NULL` terminator is reached (*i.e.,* `NULL` is not within the bounds).

Ensuring Spatial Memory Safety. The Checked C compiler instruments loads and stores of checked pointers to confirm the pointer is non-null, and additionally the access to arr and $ntarr$ pointers is within their specified bounds. For example, in the code `if (n>0)a[n-1] = ...` the write is via address $\alpha =$ `a + sizeof(int)` $\times$ `(n-1)`. If the bounds of `a` are `count(u)`, the inserted check confirms `a` $\leq \alpha <$ `a + sizeof(int)` $\times$ `u` prior to dereference. Failed checks throw an exception. Oftentimes, inserted checks can be optimized away by LLVM resulting in almost no runtime overhead [49].

Backward Compatibility. Checked C is backward compatible with legacy C as all legacy code will type-check and compile. However, the compiler adds the aforementioned spatial safety checks to only checked pointers. The spatial safety guarantee is partial when the code is not fully ported.

No Safety Against Unchecked Pointers. A partially annotated program can still enjoy spatial safety only if checked pointers do not communicate with any unchecked ones. For instance, in the example below, there are no spatial safety violations in the function `func` as it uses only checked pointers. However, the other unconverted code regions (or unsafe regions) can affect pointers in safe regions and violate certain assumptions leading to vulnerabilities, as demonstrated by cross-language attacks [50].

1) Converting C to Checked C: The safety guarantees of Checked C come with certain restrictions. For instance, as shown below, Checked C programs cannot use address-taken variables in a bounds expression as the bounds relations may not hold because of possible modifications through pointers.

```
...
_Array_ptr<int> p : count (n) = NULL;
✖..,&n,.
```

Consequently, converting existing C programs to Checked C might require refactoring, *e.g.,* eliminate `&n` from the program above without changing its functionality. This might require considerable effort [49] depending on the program's complexity. Recently, Machiry *et al.,* developed 3C [32] that tries to automatically convert a program to Checked C by adding appropriate pointer annotations. However, as described in 3C, completely automated conversion is *infeasible*, and it requires the developer to convert some code regions manually.

2) CHECKCFLEX (C_{box}): This is not a separate isolation mechanism but rather a technique to guarantee additional safety to Checked C. Here, we use CHECKCFLEX, our integration of TYPEFLEXER with Checked C types. Specifically, in a partially converted code, we consider all Checked C code and pointers as c region and unchecked code and pointers as u region. The u region can be isolated by any of the above isolation mechanisms.

Security Guarantees The use of Checked C types in c region guarantees that there will be no spatial safety violation in it. The use of isolation ensures that unchecked pointers (now u region pointers) will not affect Checked C pointers (*i.e.,* c region pointers), preventing cross-language attacks as explained in Section G.

Overhead Aspects Since C_{box} uses Checked-C (T(Checked C)) in conjunction with either I_{mem}, S_{all}, or S_{all} isolation mechanisms (T(Iso)), Overhead of C_{box} can be represented as:

$$T(C_{box}) = T(\text{Checked C}) + T(Iso) \qquad (4)$$

3) C_{box} Overhead: As explained in Section V-C, C_{box} is same as S_{all}, but the c region code will have Checked C annotations. We have tested this on only one program *i.e.,* `Parsons`, because of the manual effort [32] in adding Checked C annotations.

We observed that the overhead of C_{box} is the same as that of S_{all}, indicating that Checked C annotations in c region do not have any performance overhead. This is in line with the observations made by previous work [49], which demonstrate that Checked C does not have any performance overhead.

H. Well-formedness and Metatheories

$$m \vdash \texttt{int} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \texttt{ptr}^{\xi}\ [\beta\ \tau]_{\kappa}}$$

$$m \wedge \texttt{u} = \texttt{u} \qquad \texttt{c} \wedge m = m \qquad m_1 \wedge m_2 = m_2 \wedge m_1$$

Fig. 10: Well-formedness for Nested Pointers

In our formalism, every c mode constant pointer requires a validity check to make sure that heap is consistant. Here, we require a static verification procedure for validating a literal pointer, which is similar to the dynamic verification process in Section IV.

The verification process $\mathcal{H} \vdash n : \tau$ checks (Figure 11) validates the literal $n{:}\tau$, where $\mathcal{H}$ is the initial heap that the literal resides. For a c mode pointer, we verify that its pointer address is well-defined in $\mathcal{H}$, as well as its element value is recursively well-defined. In addition, we do not verify any u pointer, but make sure that c pointers are well-defined.

Now, we show some Well-formedness and consistency definitions that are required in showing meta-theories. The type preservation theorem relies on several *well-formedness*:

$$\mathcal{H} \vdash n : \texttt{int} \qquad \mathcal{H} \vdash 0 : \texttt{ptr}^{m'}\ \tau \qquad \mathcal{H} \vdash n : \texttt{ptr}^{\texttt{u}}\ \tau$$

$$\frac{\mathcal{H} \vdash \mathcal{H}(n) : \tau}{\mathcal{H} \vdash n : \texttt{ptr}^{\texttt{c}}\ \tau}$$

Fig. 11: Verification/Type Rules for Constants

Definition 2 (Type Environment Well-formedness): A type environment Γ is well-formed if every variable mentioned as type bounds in Γ are bounded by `int` typed variables in Γ.

Definition 3 (Heap Well-formedness): A heap $\mathcal{H}$ is well-formed if (i) $\mathcal{H}(0)$ is undefined, and (ii) for all $n : \tau$ in the range of $\mathcal{H}$, type τ contains no free variables.

Definition 4 (Stack Well-formedness): A stack snapshot φ is well-formed if for all $n{:}\tau$ in the range of φ, type τ contains no free variables.

We also need to introduce a notion of *consistency*, relating heap environments before and after a reduction step, and type environments, predicate sets, and stack snapshots together.

Definition 5 (Stack Consistency): A type environment Γ and stack snapshot φ are consistent—written $\Gamma \vdash \varphi$—if for every variable x, $\Gamma(x) = \tau$ implies that $\varphi(x) = n{:}\tau$.

Definition 6 (Stack-Heap Consistency): A stack snapshot φ is consistent with heap $\mathcal{H}$—written $\mathcal{H} \vdash \varphi$—if for every variable x, $\varphi(x) = n{:}\tau$ implies $\mathcal{H} \vdash n : \tau$.

Definition 7 (Checked Heap-Heap Consistency): A heap $\mathcal{H}'$ is consistent with $\mathcal{H}$—written $\mathcal{H} \rhd \mathcal{H}'$—if for every constant n, $\mathcal{H} \vdash n : \tau$ implies $\mathcal{H}' \vdash n : \tau$.

Here, we discuss our main meta-theoretic results for COREFLEXER: non-exposure, type preservation, and clean separation. These proofs have been conducted in our Coq model.

We first show the non-exposure theorem, where code in u region cannot access a valid c pointer address. By accessing, we refer to the dereference, assignment, `malloc`, and `free` operations.

Theorem 4 (Non-Exposure): For any COREFLEXER program e, heap $\mathcal{H}$, stack φ, type environment Γ, and variable predicate set Θ that are all are well-formed and well typed ($\Gamma \vdash_m e : \tau$ for some τ), if there exists φ', Θ', $\mathcal{H}'$ and e', such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{\texttt{u}} (\varphi', \Theta', \mathcal{H}', e')$ and $e = E[e']$ and $mode(E) = \texttt{u}$, thus, e' does not access a c pointer.

The non-exposure theorem prevents two vulnerabilities. First, it prevents that the misuse of pointers in u region do not affect the c region execution, such as 🐞 in Figure 1. Second, it represents how trampoline functions are organized to support the callback mechanism in TYPEFLEXER.

The COREFLEXER does not have type progress, referring to that a well-typed C program might not make a move. However, COREFLEXER has type preservation in c code region, which relies on several *consistency* definitions given in Appendix H.

Theorem 5 (Type Preservation): For any COREFLEXER program e, heap $\mathcal{H}$, stack φ, type environment Γ being well-formed and consistent ($\Gamma \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma \vdash_{\texttt{c}} e : \tau$ for some τ), if there exists φ', Θ' $\mathcal{H}'$ and e', such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_m (\varphi', \Theta', \mathcal{H}', e')$, then $\mathcal{H}'$ is consistent with $\mathcal{H}$ ($\mathcal{H} \triangleright \mathcal{H}'$) and there exists Γ' and τ' that are well formed, consistent ($\Gamma' \vdash \varphi'$ and $\mathcal{H}' \vdash \varphi'$) and well typed ($\Gamma' \vdash_{\mathtt{c}} e : \tau$).

We define a state to be *stuck* and *critically stuck* below, and then show our main result, *clean separation*, which suggests that a well-typed program can never be critically stuck in $\mathtt{c}$ code regions.

Definition 8 (Critically Stuck): For a program e and environment tuple $(\varphi, \Theta, \mathcal{H}, e)$, we define it to be stuck as that there is no transition tuple $(\varphi', \Theta', \mathcal{H}', r)$, such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{\mathtt{c}} (\varphi', \Theta', \mathcal{H}', r)$; it is critically stuck, when $(\varphi, \Theta, \mathcal{H}, e)$ is stuck, and e is of the three situations:

- $e = *\ n{:}\mathtt{ptr}^{\mathtt{u}}\ \tau$.
- $e = *\ n{:}\mathtt{ptr}^{\mathtt{u}}\ \tau = n'{:}\tau'$.

Theorem 6 (Clean Separation): For any CoreFlexer program e, heap $\mathcal{H}$, stack φ, type environment Γ, and set Θ that are well-formed and consistent ($\Gamma \vdash \varphi$ and $\mathcal{H} \vdash \varphi$), if e is type-preserved ($\varphi \vdash_{\mathtt{c}} e : \tau$ for some τ) and there exists φ_i, Θ_i, $\mathcal{H}_i$, e_i, and m_i for $i \in [1, k]$, such that $(\varphi, \Theta, \mathcal{H}, e) \longrightarrow_{m_1} (\varphi_1, \Theta_1, \mathcal{H}_1, e_1) \longrightarrow_{m_2} \ldots \longrightarrow_{m_k} (\varphi_k, \Theta_k, \mathcal{H}_k, r)$, then r can never be *critically stuck*.

Clean separation suggests that $\mathtt{c}$ and $\mathtt{u}$ regions are completely separated, because the tainted pointers, the only types of pointers that communicate the $\mathtt{c}$ and $\mathtt{u}$ regions, do not cause any problem in $\mathtt{c}$ regions. The 🐞 in Figure 1 is dynamically caught through the dynamic checks in TypeFlexer and it is included in the non-exposure theorem. The additional guarantee the clean separation provides is to ensure that even if a $\mathtt{u}$ mode pointer is freed in $\mathtt{u}$ region, when we access it in $\mathtt{c}$ region, our compiler can discover the error, explained in Section V-A as the use-after-free dynamic check.

I. Additional Program evaluations

Here, we provide the description of additional program evaluations.

parsons. Parsons is annotated by considering the JSON data as tainted. Consequently, we mark all the data structures that store the JSON data as tainted. Consequently, we modify all the functions that process these data structures to operate on tainted values. We made annotations on parsons to support I_{mem}, S_{mem}, S_{all}, and C_{box} partitioning mechanisms. Benchmarks for both of these forks are recorded using the mean difference between the TypeFlexer and each of these variants when executing 10 consecutive iterations of the test suite. Since I_{mem} internally uses hoard allocators, our benchmarking for this is made in comparison to a slightly modified version of parsons that internally use hoard allocators. This is done to nullify any unfair advantage offered by hoard allocators.

LibPNG. TypeFlexer changes for libPNG is narrow in scope and begin with the encapsulation CVE-2018-144550 and a buffer overflow in compare_read(). However, we also annotate sections of Lib-png that involve reading, writing, and image processing (interlace, intrapixel, etc) on user-input image data as tainted. That is, rows of image bytes are read into tainted pointers and the taintedness for the row_bytes is propagated throughout the program. All our changes extend to the png2pnm and pnm2png executables. For evaluation, we run png2pnm on a 35MB large image of size 5184x3456 to convert it to a pnm file. Similarly, we run pnm2png to convert the output from above (52MB) to png. We validate our results by ensuring a successful loop alongside all tests from libpng's pre-defined test suite. We repeat this experiment over 10 iterations and take the mean value as the record result.

MicroHTTPD. MicroHTTPD demonstrates the practical difficulties in converting a program to TypeFlexer. Our conversion for this program was aimed at sandboxing memory vulnerabilities CVE-2021-3466 and CVE-2013-7039. CVE-2021-3466 is described as a vulnerability from a buffer overflow that occurs in an unguarded ”memcpy” which copies data into a structure pointer (struct MHD_PostProcessor pp) which is type-casted to a char buffer (char *kbuf = (char *) &pp[1]). Our changes would require making the ”memcpy” safe by marking this pointer as tainted. However, this would either require marshaling the data pointed by this structure (and its sub-structure pointer members) pointer or would require marking every reference to this structure pointer as tainted, which in turn requires every pointer member of this structure to be tainted. Marshaling data between structure pointers is not easy and demands substantial marshaling code due to the spatial non-contiguity of its pointer members unlike a char*. This did not align with our conversion goals which were aimed at making minimal changes. Consequently, the above CVE stands un-handled by TypeFlexer. Our changes for CVE-2013-7039 involve marking the user input data arguments of this function as tainted pointers and in the interests of seeking minimal conversion changes, we do not propagate the taintedness on these functions. Following up on the chronological impossibility of sandboxing bugs before they are discovered and the general programmer intuition, we moved several core internal functions (like MHD_str_pct_decode_strict_() and MHD_http_unescape()) to execute within the sandbox.

ProFTPD:. TypeFlexer changes for ProFTPD were limited and made to exactly encapsulate CVE-2010-4221 by marking the user input to the unsafe function ”pr_netio_telnet_gets()” as tainted. Although we propagate the taintedness of the above function's argument, run-time overhead measured by following the methodology in VII was minimal as the data-flow graph for the tainted pointer was small and thereby, required fewer pointers to be annotated.

UFTPD. TypeFlexer changes for UFTPD were aimed at sandboxing CVE-2020-14149 and CVE-2020-5204. CVE-2020-14149 was recorded as a NULL pointer dereference in the handle_CWD() which could have led to a DoS in versions before 2.12, thereby, requiring us to sandbox this function. CVE-2020-5204 was recorded as a buffer overflow vulnerability in the handle_PORT() due to sprintf() which also required us to sandbox this function.