ConfLLVM: Compiler-Based Information Flow Control in Low-Level Code

Abstract—**We present a compiler-based scheme for information flow control in low-level applications (e.g. those written in C) in the presence of an active adversary. In our scheme, the programmer marks private data by writing lightweight annotations on the top-level definitions in the source code. The compiler then uses a combination of static dataflow analysis and runtime instrumentation to prevent data leaks even in the presence of low-level attacks. To keep the overheads of the instrumentation low, the compiler uses a novel memory layout and a *taint-aware* form of control flow integrity.**

We formalize our scheme and prove its security. We have also implemented our scheme within the LLVM compiler and evaluated it on the CPU-intensive SPEC micro-benchmarks, and on larger, real-world applications, including the NGINX webserver and the OpenLDAP directory server. We find that performance overheads introduced by our instrumentation are moderate (average 12% on SPEC), and the programmer effort to port the applications is minimal.

I. Introduction

We consider the problem of information flow control [23] for applications written in low-level languages like C. In this setting, the application has (conceptually) separate *private* and *public* data, and the goal is to ensure that the private data does not directly leak to a public channel – the application must declassify (e.g. encrypt) the private data before sending it out. Common examples of such applications include web servers that compute with sensitive data such as passwords and private files, medical software that work with private medical records, database query engines, etc.

Information flow control for low-level code is a challenging problem because leaks can occur due to several reasons. First, the program may have bugs in its logic that can inadvertently leak private data (e.g. `network_send (passwd)`). Second, low-level languages like C do not provide memory safety. As a result, contents of private buffers can overflow to public buffers, ultimately leaking via public channels. The Heartbleed bug in OpenSSL [5] is a prominent example of a buffer overflow vulnerability that can be exploited this way to leak sensitive data. Finally, low-level languages do not provide control flow integrity (CFI), making it possible for an active attacker to craft an exploit, hijack control of the application, and steal private data [10], [4], [1], [47], [41], [2].

Existing approaches for information flow control can be broadly classified into static and dynamic. Static approaches, e.g. information flow type-systems [27], [42], require the underlying language to be memory- and type-safe, making them unsuitable for C directly. As an alternative, one could use these techniques with safe dialects of C, that provide memory-safety but no information flow control (e.g. CCured [36], Deputy [20] and SoftBound [35]). These dialects also (a) require additional annotations and program restructuring that cause significant programming overhead [36], [33], (b) are not always backward compatible with legacy code, and (c) have prohibitive runtime overhead making them a non-starter in practical applications (see Section X).

Dynamic approaches track the *taint* (private or public) associated with each memory location through runtime instrumentation. When data is written to public channels, the runtime checks ensure that the data's taint is actually public. While dynamic approaches are directly applicable to C, they have substantial runtime overheads [44], [31], [53], [21]. For instance, TaintCheck [38] reports up to 37× overhead for CPU-bound applications.

As a result, an efficient, end-to-end solution for information flow control for low-level code remains elusive.

In this paper we present an efficient, compiler-based information flow control scheme for low-level code, focusing on *explicit flows*. Section II describes our threat model in detail. Ours is a hybrid scheme, combining static and dynamic techniques, based on two key insights specific to low-level code. First, complete memory safety and perfect CFI are neither sufficient nor necessary for preventing leaks. Instead, we use a harmonious combination of dataflow analysis, memory isolation, and a *taint-aware* form of CFI for information flow control. Second, the runtime overhead can be reduced by carefully controlling how data is laid out in memory. We accompany our scheme with formal security proofs and a thorough empirical evaluation to show that it can be used for real applications with overheads that are an order-of-magnitude less than purely dynamic approaches. Below, we highlight the main components of our scheme.

Annotating private data. We require the programmer to add `private` type annotations only to top-level function signatures and globals' definitions to mark private data. The programmer is free to use all of the C language, including pointer casts, aliasing, indirect calls, varargs, variable length arrays, etc.

Flow analysis and runtime instrumentation. Our compiler performs standard dataflow analysis, statically propagating taint from global variables and function arguments that have been marked `private` by the programmer, to detect any information leaks. However, in the presence of pointer casts, memory errors, or low-level attacks, it is possible that a pointer assumed public actually points to a private value at runtime (or viceversa). This can cause an information flow leak. To prevent this, our compiler *instruments* memory reads and writes with runtime checks to ensure that they read (or write) public or private data as anticipated during compilation. Pleasantly, because of these checks, we don't need any static alias analysis. However, such checks *could* have high overhead.

Novel memory layout. To reduce this overhead, we use a novel, compiler-enforced memory layout. Our compiler partitions the program's virtual address space into a contiguous public region and a disjoint, contiguous private region, each with its own stack and heap. Runtime checks for public or private are then simple, efficient range checks on pointer values. We describe two partitioning schemes, one based on the Intel MPX ISA extension [6], and the other based on segment registers (Section III). Our runtime checks do not enforce full memory safety: A private (public) pointer may point outside its expected object but our checks ensure that it always dereferences *somewhere* in the private (public) region. These lightweight checks suffice for information flow control.

Information flow-aware CFI. Similar to memory safety, we observe that perfect CFI is neither necessary nor sufficient for information flow. We introduce an efficient, taint-aware CFI scheme. Our compiler instruments the targets of indirect jumps with *magic sequences* that summarize the output of the static dataflow analysis at those points. At the source of the indirect jumps, a runtime check ensures that the magic sequence at the target is taint-consistent with the output of the static dataflow analysis at the source (Section IV).

Trusted components. For selective declassifications, as required for most practical applications, we allow the programmer to re-factor *trusted* declassification functions into a separate component, which we call $\mathcal{T}$. The remaining untrusted application, in contrast, is called $\mathcal{U}$. Code in $\mathcal{T}$ is not subject to any flow checks and can be compiled using a vanilla compiler. It can access all of $\mathcal{U}$'s memory, specifically, it can perform declassification by copying data from $\mathcal{U}$'s private region to $\mathcal{U}$'s public region (e.g. after encrypting the data). $\mathcal{T}$ has its own separate stack and heap, and we re-use the range checks on memory accesses in $\mathcal{U}$ to prevent them from reading or writing to $\mathcal{T}$'s memory. We give an example and general guidelines for refactoring $\mathcal{U}$ and $\mathcal{T}$ in Section II.

All these ingredients combined together yield an efficient information flow control scheme, with formal guarantees, as we summarize below:

Formal guarantees. We have formalized our scheme using a core language of memory instructions (load and store) and control flow instructions (goto, conditional, and direct and indirect function call). We prove a termination-insensitive non-interference theorem for $\mathcal{U}$, assuming that the $\mathcal{T}$ functions it calls are non-interferent. In other words, we prove that if two public-equivalent configurations of $\mathcal{U}$ take a step each, then the resulting configurations are also public-equivalent. Our formal model shows the impossibility of sensitive data leaks even in the presence of features like aliasing and casting, and low-level vulnerabilities such as buffer overflows and ROP attacks (Section VI).

Implementation. We have implemented our compiler, CONFLLVM, within the LLVM framework [32]. CONFLLVM performs dataflow analysis on annotated $\mathcal{U}$ code, and compiles

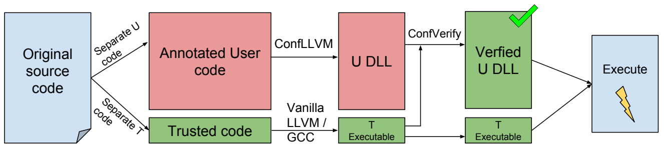


FIGURE 1: WORKFLOW OF OUR SCHEME AND TOOLCHAIN

it with the memory partitioning scheme and the runtime checks mentioned above.

We have also developed a lightweight static verifier CONFVERIFY to check that the binary output by our compiler indeed has all the required instrumentation. CONFVERIFY disassembles the binary (using hints provided by CONFLLVM) and performs dataflow analysis to ensure that there are no data leaks. Importantly, this allows us to remove the compiler from our Trusted Computing Base (Sections V and VII).

Figure 1 shows the high-level workflow of our scheme.

Evaluation. We have evaluated CONFLLVM on the standard CPU-intensive SPEC benchmarks, and several applications, including the NGINX webserver and the OpenLDAP directory server. We find that performance overheads introduced by our instrumentation are moderate, and the programmer effort for annotations and refactoring is small (Section VIII).

II. OVERVIEW

Threat model. We consider C applications that work with both private and public data. Applications interact with the external world using the network, disk and other channels. They communicate public data in clear, but want to protect the confidentiality of the private data by, for example, encrypting it before sending it out. However, the application could have logical or memory errors, or exploitable vulnerabilities that may cause private data to be leaked out in clear.

The attacker actively interacts with the application by sending inputs that are crafted to trigger any bugs in the application. The attacker can also observe all the external communication of the application. Our goal is to prevent the private data of the application from leaking out in clear. Specifically, we address *explicit* information flow: any data directly derived from private data is also treated as private. While this addresses most commonly occuring exploits [38], optionally, our scheme can be used in a stricter mode where it disallows branching on private data, thereby preventing implicit flows too [1]. Side-channels (such as execution time and memory-access patterns) are outside the scope of this work.

Our scheme can also be used for integrity protection in a setting where an application computes over trusted and untrusted data [15]. Any data (explicitly) derived from untrusted inputs cannot be supplied to a sink that expects trusted data (Section VIII-B shows an example).

Example application. Consider the code for a web server in Figure 2. The server receives requests from the user (`main`:7), where the request contains the username and a file name (both in clear text), and the encrypted user password. The server decrypts the password and calls the `handleReq` helper routine

[1]We ran all our experiments (Section VIII) using this stricter mode.

```
   void handleReq (char *uname, char *upasswd, char *fname,
2                  char *out, int out_size)
   {
4    char passwd[SIZE], fcontents[SIZE];
     read_password (uname, passwd, SIZE);
6    if(!(authenticate (uname, upasswd, passwd))) {
       return;
8    }
     //inadvertently copying the password to the log file
10   send(log_file, passwd, SIZE);

12   read_file(fname, fcontents, SIZE);
     //(out_size > SIZE) can leak passwd to out
14   memcpy(out, fcontents, out_size);
     //a bug in the fmt string can print stack contents
16   sprintf(out + SIZE, fmt, "Request complete");
   }
```

```
1  #define SIZE 512

3  int main (int argc, char **argv)
   {
5    ... //variable declarations
     while (1) {
7      n = recv(fd, buf, buf_size);
       parse(buf, uname, upasswd_enc, fname);
9      decrypt(upasswd_enc, upasswd);
       handleReq(uname, upasswd, fname, out,
11               size);
       format(out, size, buf, buf_Size);
13     send(fd, buf, buf_size);
     }
15 }
```

FIGURE 2: REQUEST HANDLING CODE FOR A WEB SERVER

that copies the (public) file contents into the `out` buffer. The server finally prepares the formatted response (`format`), and sends the response (`buf`), in clear, to the user.

The `handleReq` function allocates two local buffers, `passwd` and `fcontents` (`handleReq`:4). It reads the actual user password (e.g., from a database) into `passwd`, and authenticates the user. On successful authentication, it reads the file contents into `fcontents`, copies them to the `out` buffer, and appends a message to it signalling the completion of the request.

The code has several bugs that can cause it to leak the user password. First, at line 10, the programmer leaks the clear-text password to a log file by mistake. Second, at line 14, `memcpy` will read `out_size` bytes from `fcontents` and copy them to `out`. If `out_size` is greater than `SIZE`, this can cause the contents of `passwd` to be copied to `out` because an overflow past `fcontents` would go into the `passwd` buffer. Second, if the format string `fmt` in the `sprintf` call at line 16 contains extra formatting directives, it can print stack contents into `out` ([49]). The situation is worse if `out_size` or `fmt` can be influenced by the attacker.

Our goal is to prevent such vulnerabilities from leaking out sensitive application data. Below we discuss the three main components of our approach.

a) Partitioning the application into $\mathcal{U}$ and $\mathcal{T}$: The programmer begins by partitioning the application into *untrusted* and *trusted* components, $\mathcal{U}$ and $\mathcal{T}$ respectively. $\mathcal{T}$ is a library that provides a small set of trusted routines to $\mathcal{U}$ such as: (1) communication interfaces to the external world (e.g. networking, I/O, and other system calls), (2) cryptographic primitives and other trusted declassification functions for valid private-to-public conversions. All the other code becomes part of $\mathcal{U}$.

A good practice is to contain most of the application logic to $\mathcal{U}$ and limit $\mathcal{T}$ to a library of generic routines that can be hardened over time, possibly even verified manually [50].

In the web server example from Figure 2, $\mathcal{T}$ would consist of: `recv`, `send`, `read_file` (network, I/O), `decrypt` (cryptographic primitive), and `read_passwd` (source of sensitive data). The remaining web server code (`parse`, `format`, and even `sprintf` and `memcpy`) remains in $\mathcal{U}$ and is not trusted.

b) Partitioning of $\mathcal{U}$ memory by CONFLLVM*:* The programmer compiles $\mathcal{T}$ with a compiler of her choice (or uses existing binaries), and $\mathcal{U}$ with our compiler CONFLLVM.

CONFLLVM partitions the memory of $\mathcal{U}$ into two regions, one for public data and one for private data, with each region having its own stack and heap. To help the compiler lay out data in these regions, the programmer can annotate the sensitive data in $\mathcal{U}$ using the `private` type qualifier [24] supported by CONFLLVM. We require the programmer to annotate private data *only* in top-level definitions, i.e., globals, function signatures, and `struct` definitions, and in the prototypes of all functions exported by $\mathcal{T}$ to $\mathcal{U}$. CONFLLVM infers annotations for locals (Section V). The annotations for $\mathcal{T}$ functions are trusted (since $\mathcal{T}$ is not compiled with CONFLLVM), as opposed to the annotations in $\mathcal{U}$ which are enforced by the compiler – an incorrect $\mathcal{U}$ annotation will either cause a static error, or crash the program at runtime, but the confidentiality of the private data will not be compromised.

In our example, the untrusted annotated signatures in $\mathcal{U}$ are:

```
void handleReq(char *uname, private char *upasswd,
               char *fname, char *out, int out_sz);
int authenticate(char *uname, private char *upass,
                 private char *pass);
```

CONFLLVM can automatically infer that, for example, `passwd` is a `private` buffer. The trusted annotated signatures of $\mathcal{T}$ against which CONFLLVM compiles $\mathcal{U}$ are the following:

```
int recv(int fd, char *buf, int buf_size);
int send(int fd, char *buf, int buf_size);
void decrypt(char *ciphertxt, private char *data);
void read_passwd(char *uname, private char *pass,
                 int size);
```

CONFLLVM raises a compile-time error flagging the bug at line 10. Once the bug is fixed by the programmer (e.g. by removing the line), CONFLLVM compiles the program and lays out the stack and heap data in their corresponding regions (Section III), with runtime checks to enforce the annotations in the presence of imprecision caused by casting, aliasing, etc. The other two leaks are then prevented at runtime by failed checks.

c) Runtime checks: The runtime checks ensure that, (a) the pointers belong to their annotated or inferred regions (e.g. a `private char *` actually points to the private region), (b) $\mathcal{U}$ does not read or write beyond its own memory (i.e., it does not read or write to $\mathcal{T}$ memory), and (c) $\mathcal{U}$ follows a *taint-aware* form of CFI that prevents circumvention of the checks and prevents data leaks due to control-flow attacks.

In addition, $\mathcal{T}$ functions must appropriately check their arguments to ensure that the data passed by $\mathcal{U}$ has the correct sensitivity label. For example, the `read_passwd` function would check that the range [`pass`, `pass+size-1`] falls inside the private memory segment of $\mathcal{U}$. (Note that this is different from complete memory safety; $\mathcal{T}$ need not know the allocation size of the `passwd` buffer.)

Our scheme requires that the public and private partitions are not executable (Section VII). We do not yet support dynamic code generation.

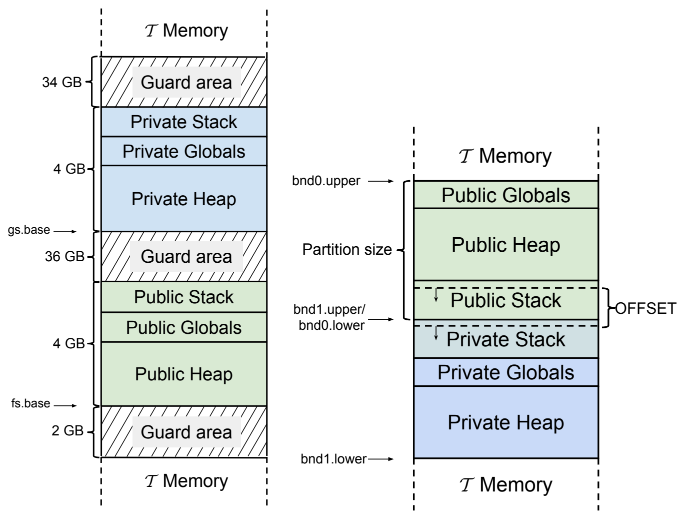
Trusted Computing Base (TCB). We have also designed and implemented a static verifier, ConfVerify, to confirm that a binary output by ConfLLVM has enough checks in place to guarantee confidentiality (Section V). ConfVerify guards against bugs in the compiler.

Our TCB, and thus the security of our scheme, does not depend on the untrusted application code $\mathcal{U}$ or the compiler. We trust the library code $\mathcal{T}$ and the static verifier. We also trust the operating system and other components running at a privileged level (although, we can potentially make use of trusted hardware like Intel SGX to isolate the memory of our application from the OS [50]).

III. Memory Partitioning Schemes

ConfLLVM uses the programmer-supplied annotations, and with the help of type inference, statically determines the taint of each memory access (SectionV), i.e., for every memory load and store, it knows statically if the address contains private or public data. It is possible for the type-inference to detect a problem (for instance, when a variable holding private data is passed to a method expecting a public argument), in which case, a type error is reported back to the programmer. On successful inference, no data leaks are guaranteed, *provided* the programmer-supplied annotations on $\mathcal{T}$ exported functions are correct. We have developed two different schemes for enforcing annotation correctness at runtime, each of which relies on a careful layout of $\mathcal{U}$ memory. The key idea in each scheme is to contain all private and all public data to their own respective contiguous regions of memory. The rest of this section outlines these two techniques and discusses their relative pros and cons.

MPX scheme. This scheme relies on the Intel MPX ISA extension [6] and uses the memory layout shown in Figure 3b. The memory is partitioned into a public region and a private region, each with its own heap, stack and global segments. The ranges of these regions are determined by the values stored in the MPX bound registers `bnd0` and `bnd1`, respectively, but they must be contiguous to each other (`bnd0.lower == bnd1.upper`). The user selects the maximum stack size `OFFSET` at compile time (at most $2^{31}-1$). The scheme maintains the public and private stacks in lock-step: their respective top-of-stack are always at offset `OFFSET` to



(a) Segment scheme (b) MPX scheme

Figure 3: Memory layout of $\mathcal{U}$

each other. The concrete values stored in `bnd0` and `bnd1` can be determined at load time (Section VII).

Consider the procedure in Figure 4a. The generated (unoptimized) assembly under the MPX scheme (using virtual registers for simplicity) is shown in Figure 4c. ConfLLVM automatically infers that `x` is a `private int` and places it on the private stack, whereas `y` is kept on the public stack. The stack pointer `rsp` points to the top of the public stack. Because the two stacks are kept at constant `OFFSET` to each other, `x` is accessed simply as `rsp+4+OFFSET`. Each memory access is preceded with MPX instructions (`bndcu` and `bndcl`) that check their first argument against the (upper and lower) bounds of their second argument.

Segmentation scheme. x64 memory operands are in the form $[base + index * scale + displacement]$, where *base* and *index* are 64-bit unsigned registers, *scale* is a constant with maximum value of 8, and *displacement* is a 32-bit signed constant. The architecture also provides two segment registers `fs` and `gs` for the *base* address computation, i.e. `fs`:*base* simply adds `fs` to the *base* value[2].

We use these segment registers to store the lower bounds of the public and private memory regions, respectively, and follow the memory layout shown in Figure 3a. The public and private regions are separated by (at least) 36GB of guard space (unmapped pages that cause a fault when accessed). The guard sizes are chosen so that any memory operand whose *base* is prefixed with `fs` cannot escape the public segment, and any memory operand prefixed with `gs` cannot escape the private segment.

The segments are each aligned to a 4GB boundary. The usable space within each segment is also 4GB. We access the *base* address stored in a 64-bit register, say a private value stored in `rax`, as `fs+eax`, where `eax` is the lower 32 bits of `rax`. Thus, in `fs+eax`, the lower 32 bits come from `eax` and the upper 32 bits come from `fs` (because `fs` is 4GB aligned). This

[2]The segment registers also carry a *limit* but it is unused in x64.

```
private int bar (private int *p, int *q)
{
  int x = *p;
  int y = *q;
  return x + y;
}
```

(a) A sample $\mathcal{U}$ procedure

```
;argument registers p = r1, p = r2

sub rsp, 16             ;rsp = rsp - 16
r3 = load gs:[e1]       ;r3 = *p
store gs:[esp+4], r3    ;x = r3
r4 = load fs:[e2]       ;r4 = *q
store fs:[esp+8], r4    ;y = r4
r5 = load gs:[esp+4]    ;r5 = x
r6 = load fs:[esp+8]    ;r6 = y
r7 = r5 + r6
add rsp, 16             ;rsp = rsp + 16
ret r7
```

(b) Assembly code under segment scheme

```
;argument registers p = r1, q = r2
;stack offsets from rsp: x: 4, y: 8
sub rsp, 16                  ;rsp = rsp - 16
bndcu [r1], bnd1             ;MPX instructions to check that-
bndcl [r1], bnd1             ;-r1 points to private region
r3 = load [r1]               ;r3 = *p
bndcu [rsp+4+OFFSET], bnd1   ;check that rsp+4+OFFSET-
bndcl [rsp+4+OFFSET], bnd1   ;-points to private region
store [rsp+4+OFFSET], r3     ;x = r3
bndcu [r2], bnd0             ;check that r2 points to-
bndcl [r2], bnd0             ;-the public region
r4 = load [r2]               ;r4 = *q
bndcu [rsp+8], bnd0          ;check that rsp+8 points to-
bndcl [rsp+8], bnd0          ;-the public region
store [rsp+8], r4            ;y = r4
bndcu [rsp+4+OFFSET], bnd1
bndcl [rsp+4+OFFSET], bnd1
r5 = load [rsp+4+OFFSET]     ;r5 = x
bndcu [rsp+8], bnd0
bndcl [rsp+8], bnd0
r6 = load [rsp+8]            ;r6 = y
r7 = r5 + r6
add rsp, 16                  ;rsp = rsp + 16
ret r7
```

(c) Assembly code under MPX scheme

FIGURE 4: THE (UNOPTIMIZED) ASSEMBLY GENERATED BY CONFLLVM FOR AN EXAMPLE PROCEDURE.

additionally implies that the maximum offset within a segment that $\mathcal{U}$ can access is 38GB $(4 + 4 * 8 + 2)$. This is rounded up to 40GB for 4GB alignment, with 4GB of usable space and 36GB of guard space. Since the *displacement* value can be negative, the maximum negative offset is 2GB, for which we have the guard space below the public segment.

The usable parts of the segments are restricted to 4GB because it is the maximum addressable size using a single 32 bit register. This restriction also ensures that we don't have to translate $\mathcal{U}$ pointers when the control is passed to $\mathcal{T}$, thus avoiding the need to change or recompile $\mathcal{T}$. Generated code for our example under this scheme is shown in Figure 4b. The figure uses the convention that $\texttt{e}_i$ (resp., `esp`) represents the lower 32 bits of the register $\texttt{r}_i$ (resp., `rsp`). The public and private stacks are still maintained in lock-step. Taking the address of a private stack variable requires extra support: the address of variable `x` in our example is `rsp+4+size`, where `size` is the total segment size (40GB).

The segmentation scheme has a lower runtime overhead than the MPX scheme as it avoids doing bound-checks. However, it restricts the segment size to 4GB.

Multi-threading support. Our schemes support multi-threading. All inserted runtime checks (including those in Section IV) are thread-safe because they check values of registers. However, we do need to introduce additional support to use thread-local storage (TLS). Typically, TLS is accessed via the segment register `gs`: the base of TLS is obtained at a constant offset from `gs`. The operating system takes care of setting `gs` on a per-thread basis. However, $\mathcal{U}$ and $\mathcal{T}$ operate in different trust domains, thus they cannot share the same TLS buffer.

We let $\mathcal{T}$ continue to use `gs` for accessing its own TLS. CONFLLVM changes the compilation of $\mathcal{U}$ to access TLS in a different way. The multiple (per-thread) stacks in $\mathcal{U}$ are all allocated inside the stack regions; the public and private stacks for each thread are still at a constant offset to each other. Each thread stack is, by default, of maximum size 1MB and its start is aligned to a 1MB boundary (configurable at compile time). We keep the per-thread TLS buffer at the beginning of the stack. $\mathcal{U}$ simply masks the lower 20-bits of `rsp` to zeros to obtain the base of the stack and access TLS.

The segment-register scheme further requires switching of the `gs` register as control transfers between $\mathcal{U}$ and $\mathcal{T}$. We use appropriate wrappers to achieve this switching, however $\mathcal{T}$ needs to reliably identify the current thread-id when called from $\mathcal{U}$ (so that $\mathcal{U}$ cannot force two different threads to use the same stack in $\mathcal{T}$). CONFLLVM achieves this by instrumenting an inlined-version of the `_chkstk` routine[3] to make sure that `rsp` does not escape its stack boundaries.

IV. INFORMATION FLOW AWARE CFI

We design a custom, information-flow aware CFI scheme to ensure that an attacker cannot alter the control flow of $\mathcal{U}$ to circumvent the instrumented checks and leak sensitive data.

Typical low-level attacks that can hijack the control flow of a program include overwriting the return address, or the targets of function pointers and indirect jumps. Existing approaches use a combination of *shadow stacks* or *stack canaries* to prevent overwriting the return address, or use fine-grained taint tracking to ensure that the value of a function pointer is not derived from user (i.e. attacker-controlled) inputs [21], [30], [50]. While these techniques may prevent certain attacks, our goal is purely to ensure confidentiality. Thus, we designed a custom taint-aware CFI scheme.

Our CFI scheme ensures that for each indirect transfer of control: (a) the target address is *some* valid jump location, i.e.,

[3]https://msdn.microsoft.com/en-us/library/ms648426.aspx

the target of an indirect call is some valid procedure entry, and the target of a return is some valid return site, (b) the register taints expected at the target address match the current register taints (e.g., when the `rax` register holds a private value then a `ret` can only go to a site that expected a `private` return value). Our scheme does not ensure, for instance, that a return matches the previous call. We use a *magic-sequence* based scheme to achieve this CFI.

CFI for function calls and returns. We follow the x64 calling convention for Windows that has 4 argument registers and one return register. Our scheme picks two bit sequences $\mathtt{M}_{\mathtt{Call}}$ and $\mathtt{M}_{\mathtt{Ret}}$ of length 59 each that appear nowhere else in $\mathcal{U}$'s binary. Each procedure in the binary is preceded with a string that consists of $\mathtt{M}_{\mathtt{Call}}$ followed by a 5-bit sequence encoding the expected taints of the 4 argument registers and the return register, as per the function signature. Similarly, each valid return site in the binary is preceded by $\mathtt{M}_{\mathtt{Ret}}$ followed by 1-bit encoding of the taint of the return value register, again according to the callee's signature. To keep the length of the sequences uniform at 64 bits, the return site taint is padded with four zeros.

Callee-save registers are also live at function entry and exit and their taints cannot be determined statically by the compiler. CONFLLVM forces their taint to be public by making the caller save and clear all the private-tainted callee-saved registers before making a `call`. All dead registers (e.g. unused argument registers and caller-saved registers at the beginning of a function) are conservatively marked `private` to avoid accidental leaks. The 64-bit instrumented sequences are collectively referred to as magic sequences. We note that our scheme can be extended easily to support other calling conventions.

Consider the following $\mathcal{U}$:

```
private int add (private int x) { return x + 1; }
private int incr (private int *p, private int x) {
  int y = add (x); *p = y; return *p; }
```

The compiled code for these functions is instrumented with magic sequences as follows. The 5 taint bits for `add` are `11111` as its argument `x` is `private`, unused argument registers are conservatively treated as `private`, and its return type is also `private`. On the other hand, the taint bits for `incr` are `01111` because its first argument is a `public` pointer (although it points to a `private` int, that is taken care of at the dereferences), second argument is `private`, unused argument registers are `private`, and the return value is also `private`. The sample instrumentation is as shown below:

```
#M_call#11111#
add:
  ...  ;assembly code for add
#M_call#01111#
incr:
  ... ;assembly code of incr
  call add
  #M_ret#00001# ;private-tainted ret with padded 0s
  ... ;assembly code for rest of incr
```

Our CFI scheme adds runtime checks using these sequences as follows. Each `ret` instruction is instrumented to instead fetch the return address and confirm that its target location has $\mathtt{M}_{\mathtt{Ret}}$ followed by the taint-bit of the return register. For our example, the return of `add` is replaced as follows:

```
#M_call#11111#
add:
  ...
  r1 = pop ;fetch return address
  r2 = #M_ret_inverted#11110# ;we use bitwise nega-
  r2 = not r2                 ;-tion of magic string
  cmp [r1], r2                ;to retain uniqueness
  jne fail                    ;of M_ret in the binary
  r1 = add r1, 8 ;skip magic sequence
  jmp r1 ;return
fail: call __debugbreak
```

For direct calls, CONFLLVM statically verifies that the register taints match between the call site and the call target. At indirect calls, the instrumentation is similar to that of a `ret`: check that the target location contains $\mathtt{M}_{\mathtt{Call}}$ followed by taint bits that match the register taints at the call site.

Indirect jumps. CONFLLVM does not generate indirect jumps in $\mathcal{U}$. Indirect jumps are mostly required for jump-table optimizations, which we currently disable. We can conceptually support them as long as the jump tables are statically known and placed in read-only memory.

The insertion of magic sequences increases code size but it makes the CFI-checking more lightweight than shadow stack schemes. The unique sequences $\mathtt{M}_{\mathtt{Call}}$ and $\mathtt{M}_{\mathtt{Ret}}$ are created post linking when both the binaries are available (Section VII).

V. IMPLEMENTATION

A. CONFLLVM

We implemented CONFLLVM as part of LLVM [32], currently targeting the Windows x64 platform.

Compiler front-end. We introduce a new type qualifier, `private`, in the language that the programmers can use to annotate types. For example, a private integer-typed variable can be declared as `private int x`, and a (public) pointer pointing to a private integer as `private int *p`. The `struct` fields inherit their *outermost* annotation from the corresponding `struct`-typed variable. For example, consider a declaration `struct st { private int *p; }`, and a variable `x` of type `struct st`. Then `x.p` inherits its qualifier from `x`: if `x` is declared as `private st x;`, then `x.p` is a private pointer pointing to a private integer. This convention ensures that despite the memory partitioning into public and private, structs are still laid out contiguously in the memory in one of the regions.

We modified the Clang [3] frontend to parse the `private` type qualifier and generate LLVM Intermediate Representation (IR) instrumented with this additional metadata. Once the IR is generated, CONFLLVM runs standard LLVM IR optimizations that are part of the LLVM toolchain. Most of the optimizations work as-is and don't require any change. Optimizations that change the metadata (e.g. remove-dead-args changes the function signatures), need to be modified. While we found that it is not much effort to modify an optimization, we chose to modify only the most important ones in order to bound our effort. Rest of the optimizations are disabled, though we anticipate that some of them can be supported with further engineering effort.

LLVM IR and type inference. After all the optimizations are run, our compiler runs a *type qualifier inference* [24] pass over the IR. This inference pass propagates the type qualifier annotations to local variables, and outputs an IR where all the intermediates are annotated with optional `private` qualifiers. The inference is implemented using the standard algorithm, where the dataflows in the program generate qualifier subtyping constraints, which are then solved using an SMT solver [22] at the backend. If the constraints are unsatisfiable, an error is reported to the user. We refer the reader to [24] for details of the algorithm.

After type inference, ConfLLVM knows the taint of each memory operand for `load` and `store` instructions. With a simple dataflow analysis [14], the compiler statically determines the taint of each register at each instruction.

Register splilling and code generation. We made the register allocator be taint-aware: when a register is to be spilled on the stack, the compiler appropriately chooses the private or the public stack depending on the taint of the register. Once the LLVM IR is lowered to machine IR, ConfLLVM emits the assembly code inserting all the checks for memory bounds and CFI.

MPX Optimizations. ConfLLVM optimizes the bounds-checking in the MPX scheme. MPX instruction operands are identical to x64 memory operands, therefore one can check bounds of a complex operand using a single instruction. However, we found that bounds-checking a register is faster than bounds-checking a memory operand (perhaps because using a memory operand requires an implicit `lea`).

ConfLLVM optimizes the checks to be on a register as much as possible. It reserves 1MB of space around the public and private regions as guard regions and eliminates the *displacement* in each memory operand if its absolute value is smaller than 2^{20}. Usually the displacement value is a small constant (for accessing structure fields or doing stack accesses) and this optimization applies to a large degree. Further, by enabling the `_chkstk` enforcement for the MPX scheme also (Section III), ConfLLVM eliminates checks on stack accesses altogether because the `rsp` value is bound to be within the public region (and `rsp+OFFSET` is bound to be within the private region).

ConfLLVM further coalesces MPX checks within a basic block. Before adding a check, it confirms if the same check was already added previously in the same block, and there are no intervening `call` instructions or subsequent modifications to its base or index registers.

B. ConfVerify

We have developed ConfVerify to check that a binary produced by ConfLLVM has the required instrumentation in place to guarantee that there are no (explicit) private data leaks. The design goal of ConfVerify is to guard against bugs in ConfLLVM; it is not a general-purpose verifier meant for arbitrary binaries. ConfVerify actually helped us catch bugs in ConfLLVM during its development.

ConfVerify is only 1500 LOC in addition to an off-the-shelf disassembler that it uses for CFG construction[4] (as compared to 5MLOC for ConfLLVM), and therefore, much easier to manually audit. It lacks all the complexities of ConfLLVM (such as register allocation, optimizations, etc.), and uses a simple dataflow analysis to check all the flows. Consequently, it provides a higher degree of assurance for the security of our scheme.

Disassembly. ConfVerify requires the unique prefixes of magic sequences (Section IV) as input and uses them to identify procedure entries in the binary. It starts disassembling the procedures and constructs their control-flow graph (CFG). ConfVerify assumes that the binary satisfies CFI, which makes it possible to reliably identify all instructions in a procedure. If the disassembly fails, the binary is rejected. Otherwise, ConfVerify checks its assumptions: that the magic sequences were indeed unique in the procedures identified and that they have enough CFI checks.

Data flow analysis and checks. Next, ConfVerify performs a dataflow analysis on the constructed CFG to determine the taints of all the registers at each instruction. It starts by picking the taint bits of the magic sequence preceding the procedure (which makes the analysis modular). It looks for MPX checks or the use of segment registers to identify the taints of memory operands; if it cannot find a check in the same basic block, the verification fails. For each `store` instruction, it checks that the taint of the destination operand matches the taint of the source register. For direct calls, it checks that the expected taints of the arguments, as encoded in the magic sequence at the callee, matches the taints of the argument registers at the callsite (this differs from ConfLLVM which uses functions signatures). For indirect control transfers (indirect calls and `ret`), ConfVerify confirms that there is a check for the magic sequence at the target site and that its taint bits match the inferred taints for registers. After a call instruction, ConfVerify picks up taint of the return register from the magic sequence (there should be one), marks other all caller-save registers as private, and callee-save registers as public (following ConfLLVM's convention).

Additional checks. ConfVerify additionally makes sure that a direct or a conditional jump can only go a location in the same procedure. ConfVerify rejects a binary that has an indirect jump, a system call, or if it modifies a segment register. ConfVerify also confirms correct usage of `_chkstk` to ensure that `rsp` is kept within stack bounds. When using the segment-register scheme, ConfVerify additionally checks that each memory operand uses the lower 32-bits of registers.

VI. Formal Analysis of ConfVerify

Conceptually, ConfVerify operates in two stages. First, it disassembles a binary, constructs its control flow graph (CFG) and re-infers the taints of registers at all program points.

[4]We use the LLVM disassembler.

$$
\begin{array}{rcl}
\mathit{Cmd} & := & \textbf{ldr}\,(\mathit{reg}, exp) \quad | \quad \textbf{str}\,(\mathit{reg}, exp) \quad | \quad \textbf{goto}\,(exp) \quad | \\
 & & \textbf{ifthenelse}\,(exp, \textbf{goto}\,(exp)\,, \textbf{goto}\,(exp)) \quad | \quad \textbf{ret} \quad | \\
 & & \textbf{call}_{\{\mathcal{U}|\mathcal{T}\}}\;\; f\,(exp^*) \quad | \quad \textbf{icall}\;\; exp\,(exp^*) \quad | \quad \textbf{assert}\,(exp^*) \\
exp & := & n \in \mathit{Val} \quad | \quad \mathit{reg} \in \mathit{Reg} \quad | \quad \Diamond_u\, exp \quad | \quad exp\, \Diamond_b\, exp \quad | \quad \&f
\end{array}
$$

TABLE I: COMMAND SYNTAX

Second, it checks that the taints at the beginning and end of every instruction are consistent with the instruction's semantics, and that other dynamic checks described in Section V-B are correctly inserted. Here, we formalize the second stage and show if a program passes those checks, then the program is secure in a formal sense. We assume that the disassembly and the reconstruction of the CFG from the first stage, both of which use an existing tool, are correct.

We model the disassembled CFG abstractly. For each function, the CFG has one bit of metadata, which indicates whether the function is part of the trusted code or the untrusted code (abstractly written $\mathcal{T}$ and $\mathcal{U}$, respectively). Additionally, there is a 64-bit magic sequence for each function, which encodes the taints of the function's arguments and its return value (see Section IV). For trusted functions (whose code CONFVERIFY does not analyze), this is all the CFG contains. For untrusted functions (whose code CONFVERIFY does analyze), there is an additional block-graph describing the code of the function. The block-graph for an untrusted function f is a tuple $G_f = \langle V_f, E \rangle$ of nodes of f and edges $E \subseteq V \times V$. The edges represent all direct control transfers. A node $\langle pc, C, \Gamma, \Gamma' \rangle \in V_f$ consists of a program counter pc, a single command (assembly instruction) C, and register taints Γ and Γ' before and after the execution of C. pc is a number, like a line number, that can be used in an indirect jump to this node. Γ and Γ' are maps from machine register ids to $\{\mathbf{H}, \mathbf{L}\}$, representing high (private) and low (public) data.

Commands C are represented in an abstract syntax shown in Table I. The auxiliary syntax of *expressions* consists of constants, which here can only be integers that may represent ordinary data or line numbers, standard binary and unary operators (ranged over by $\Diamond_b$ and $\Diamond_u$, respectively), and the special operation $\&f$ which returns the pc of the starting instruction of function f. We assume that operations used in expressions are total, i.e., expressions never get stuck.

Commands include register load $\textbf{ldr}\,(\mathit{reg}, exp)$ and store $\textbf{str}\,(\mathit{reg}, exp)$ where the input exp evaluates to a memory location, unconditional jumps $\textbf{goto}\,(exp)$, conditionals **ifthenelse**, direct function calls $\textbf{call}_{\{\mathcal{U}|\mathcal{T}\}}\; f\,(exp^*)$, indirect function calls **icall**, and return **ret**. The subscript for $\textbf{call}_{\{\mathcal{U}|\mathcal{T}\}}$ denotes if the command is used to invoke an untrusted functionality or a trusted one. An additional command **assert** models checks inserted by the compiler. The asserted expressions must be true, else the program halts in a special state $\perp$. Appendix A presents the operational semantic of the commands in Table I.

Every CFG G has a designated entry function, similar to main() in C programs.

Dynamic semantics. A program/CFG G can be evaluated in a context of a data memory $\mu : Val \to Val$, a register state $\rho : Reg \to Val$ and a program counter pc which points to a node in the program's CFG. The memory μ is actually a union of two maps $\mu_{\mathbf{L}}$ and $\mu_{\mathbf{H}}$ over disjoint domains, representing the low and the high regions of memory of our scheme. Given a fixed G, a *configuration* s of G is a record $\langle \mu, \rho, [\sigma_{\mathbf{H}} : \sigma_{\mathbf{L}}], pc \rangle$ of the entire state—the memory, register state, stack, and the program counter. We make a distinction between high ($\sigma_{\mathbf{H}}$) and low ($\sigma_{\mathbf{L}}$) stacks. We write $\langle . \rangle .i$ to project a tuple to its ith component. Thus, if $s = \langle \mu, \rho, [\sigma_{\mathbf{H}} : \sigma_{\mathbf{L}}], pc \rangle$, then $s.pc = pc$. We call a configuration s initial (final) if $s.pc$ is the first (last) node of the entry function of G.

$$
\frac{\begin{array}{c} C = \textbf{ldr}(\mathit{reg}, e) \\ \forall\; v \in \mathit{pred}(G, pc).\; v.C = \textbf{assert}(e \in \mathit{Dom}(\mu_{\ell_e})) \end{array}}{G \vdash \Gamma\{pc\}\Gamma[\mathit{reg} \mapsto \ell_e]}
$$

$$
\frac{\begin{array}{c} C = \textbf{str}(\mathit{reg}, e) \quad \Gamma \vdash \mathit{reg} : \ell_r \quad \ell_r \sqsubseteq \ell_e \\ \forall v \in \mathit{pred}(G, pc).\; v.C = \textbf{assert}(e \in \mathit{Dom}(\mu_{\ell_e})) \end{array}}{G \vdash \Gamma\{pc\}\Gamma}
$$

$$
\frac{C = \textbf{goto}\,(e) \quad \Gamma \vdash e : \ell_e \quad \ell_e \sqsubseteq \mathbf{L}}{G \vdash \Gamma\;\{pc\}\;\Gamma}
$$

$$
\frac{C = \textbf{ifthenelse}\,(e, \textbf{goto}\,(e_1)\,, \textbf{goto}\,(e_2)) \quad \Gamma \vdash e : \ell_e \quad \ell_e \sqsubseteq \mathbf{L}}{G \vdash \Gamma\;\{pc\}\;\Gamma}
$$

FIGURE 5: SELECTED TYPE RULES. C **IS THE COMMAND IN THE CFG NODE POINTED TO BY** pc**.**

Next, we define the dynamic semantics of a program as a transition relation $s \to s'$. The rules of this relation are the expected ones, so we omit them here. We only note that calls to trusted functions, whose code is not modeled in the CFG, are represented via an external relation $s \hookrightarrow_f s'$ that models the entire execution of the trusted function f and its effects on the configuration. The special transition $s \to \perp$ means that $s.pc$ contains an **assert** command, one of whose arguments evaluates to false. $\perp$ does not transition any further. Furthermore, the transition $s \to \lightning$ is used to model adversarial behaviour, for example, when the target of **icall** is not in the CFG. $\lightning$ resembles a configuration where memory locations are loaded with random values.

Security Analysis. We formalize the checks that CONFVERIFY makes via the judgment $G \vdash \Gamma\{pc\}\Gamma'$, which means that the command in the node labeled pc in the CFG G is consistent with the beginning and ending taints Γ and Γ', and that the checks from Section V-B corresponding to this command are satisfied. Selected rules for this judgment are shown in Figure 5. The function $\mathit{pred}(G, pc)$ returns the predecessors of the node labeled pc in G. For $\ell \in \{\mathbf{L}, \mathbf{H}\}$, $\textbf{assert}(e \in \mathit{Dom}(\mu_\ell))$ represents the dynamic check that e evaluates to an address in the domain of μ_ℓ, and the auxiliary judgment $\Gamma \vdash e : \ell$ means that the expression e depends only on values with secrecy ℓ or lower.

The rules are mostly self-explanatory. The first rule, which is for the command $\textbf{ldr}(\mathit{reg}, e)$, says that if reg has taint ℓ_e after the command, then on all paths leading to this command, there must be a check that whatever e evaluates to is actually pointing into μ_{ℓ_e}. The second rule, for **str**, is similar. In the rules for indirect branching, we insist that the addresses of the branch targets have the taint $\mathbf{L}$. Overall, our type system's design is inspired by the flow-sensitive information flow type system of Hunt and Sands [29]. However, the runtime checks are new to our system. (The appendix has all the rules of the type system.)

We say that a CFG G is well-typed (passes CONFVERIFY's checks), written $\vdash G$, when two conditions hold for all nodes v in (untrusted functions in) G: 1) The node (locally) satisfies the type system. Formally, $G \vdash v.\Gamma \ \{v.pc\} \ v.\Gamma'$, and 2) The ending taint of the node is consistent with the beginning taints of all its successors. Formally, for all nodes $v' \in succ(G, v.pc)$, $v.\Gamma' \sqsubseteq v'.\Gamma$, where $succ(G, pc)$ returns the successors of the node labeled pc in G. We emphasize again that only untrusted functions are checked.

Security theorem. The above checks are sufficient to prove our key theorem: If a program passes CONFVERIFY, then assuming that its trusted functions don't leak private data, the whole program does not leak private data, end-to-end. We formalize non-leakage of private data as the standard information flow property called *termination-insensitive noninterference*. Roughly, this property requires a notion of low equivalence of program configurations (of the same CFG G), written $s =_{\mathbf{L}} s'$, which allows memories of s and s' to differ only in the private region. A program is noninterfering if it *preserves* $=_{\mathbf{L}}$ for any two runs, except when one program halts safely (e.g., on a failed assertion). Intuitively, noninterference means that no information from the private part of the initial memory can leak into the public part of the final memory.

For our model, we define $s =_{\mathbf{L}} s'$ to hold if: (i) s and s' point to the same command, i.e., $s.pc = s'.pc$, (ii) the contents of their low-stacks are equal, $s.\sigma_{\mathbf{L}} = s'.\sigma_{\mathbf{L}}$, (iii) for all low memory addresses $m \in \mu_{\mathbf{L}}$, $s.\mu(m) = s'.\mu(m)$, (iv) for all registers r such that $G(s.pc).\Gamma(r) = \mathbf{L}$, $s.\rho(r) = s'.\rho(r)$.

The assumption that trusted code does not leak private data is formalized as follows.

Assumption 1. *For all s_0, s_1, s'_0 such that $s_0 =_{\mathbf{L}} s_1$, if $s_0 \hookrightarrow_f s'_0$ then $\exists s'_1.\ s_1 \hookrightarrow_f s'_1$ and $s'_0 =_{\mathbf{L}} s'_1$.*

Under this assumption on the trusted code, we can show the noninterference or security theorem. A necessary condition to show noninterference, however, is to ensure that no well-typed program can reach an ill-formed configuration.

Lemma 1. *Suppose $\vdash G$, for all configurations s of G it holds that $s \not\to^* \lightning$.*

Lemma 1 rules out possible nondeterminism caused by adversarial behaviour and allows to formalize the security theorem as follows.

Theorem 1 (Termination-insensitive noninterference). *Suppose $\vdash G$. Then, for all configurations s_0, s'_0 and s_1 of G such that $s_0 =_{\mathbf{L}} s_1$ and $s_0 \to^* s'_0$, then either $s_1 \to^* \bot$ or $\exists s'_1.\ s_1 \to^* s'_1$ and $s'_0 =_{\mathbf{L}} s'_1$.*

When s_0 and s'_0 are initial and final configurations, respectively, then s_1 and s'_1 must also be initial and final configurations, so the theorem guarantees freedom from data leaks, end-to-end (modulo assertion check failures).

VII. TOOLCHAIN

This section describes the overall flow to launch an application using our toolchain.

Compiling $\mathcal{U}$ using CONFLLVM. Recall that the only external functions in the $\mathcal{U}$ code are $\mathcal{T}$ functions; all external communication of $\mathcal{U}$ happens via $\mathcal{T}$. The $\mathcal{U}$ code is compiled with an (auto-generated) stub file, that implements each of these $\mathcal{T}$ functions as an indirect jump from a table `externals`, located at a constant position in $\mathcal{U}$ (e.g. `jmp (externals + offset)`$_i$ for the i-th function). The table `externals` is initialized with zeroes at this point, and CONFLLVM links all the $\mathcal{U}$ files to produce a $\mathcal{U}$ dll.

The $\mathcal{U}$ dll is then postprocessed to patch all the references to globals, so that they correspond to the correct (private or public) region. The globals themselves are relocated by the loader. The postprocessing pass also sets the 59-bit prefix for the magic sequences (used for CFI, Section IV). We find these sequences by generating random bit sequences and checking for uniqueness; usually a small number of iterations suffice.

Wrappers for $\mathcal{T}$ functions. For each of the functions in $\mathcal{T}$'s interface exported to $\mathcal{U}$, we write a small wrapper that: (a) performs the necessary checks for the arguments (e.g. the `send` wrapper would check that its argument buffer is contained in the public region), (b) copies arguments to $\mathcal{T}$'s stack, (c) switches `gs`, (d) switches `rsp` to $\mathcal{T}$'s stack, and (e) calls the corresponding $\mathcal{T}$ function underneath (e.g. `send` in libc). On return, it (f) switches `gs` and `rsp` back and jumps to $\mathcal{U}$ in a similar manner as our CFI return instrumentation. Additionally, the wrappers include the magic sequence similar to those in $\mathcal{U}$ so that the CFI checks in $\mathcal{U}$ do not fail when calling $\mathcal{T}$. These wrappers are compiled with the $\mathcal{T}$ dll, and the output dll exports the interface functions.

Loading the $\mathcal{U}$ and $\mathcal{T}$ dlls. When loading the $\mathcal{U}$ and $\mathcal{T}$ dlls, the loader: (1) populates the `externals` table in U with addresses of the wrapper functions in $\mathcal{T}$, (2) relocates the globals in $\mathcal{U}$ to their respective, private or public regions, (3) sets the MPX bound registers for the MPX scheme or the segment registers for the segment-register scheme, and (4) initializes the heaps and stacks in all the regions, marks them non-executable, and jumps to the `main` routine.

VIII. EVALUATION

The goal of our evaluation is three-fold: (a) Quantify the performance overheads of CONFLLVM's instrumentation, both for enforcing bounds and for enforcing CFI; (b) Quantify necessary code changes in existing applications to enable them to be compiled by CONFLLVM; (c) Check that our scheme actually stops confidentiality exploits in applications.

A. CPU benchmarks

We measured the overheads of CONFLLVM's instrumentation on the standard SPEC CPU 2006 benchmarks [11]. We treat the code of the benchmarks as untrusted (in $\mathcal{U}$), but libc functions are trusted (in $\mathcal{T}$). Since these benchmarks use no private data, we added no annotations to the benchmarks, which makes all data public by default. Nonetheless, the code emitted by CONFLLVM ensures that all memory accesses are actually in the public region, it enforces CFI, and switches stacks when calling $\mathcal{T}$ functions, so this experiment accurately captures CONFLLVM's overheads. We ran the benchmarks in the following configurations.

- **Base**: Benchmarks compiled with vanilla LLVM, with O2 optimizations. This is the baseline for evaluation.[5]
- $\textbf{Base}_{\text{OA}}$: CONFLLVM requires the use a customized memory allocator since the Windows system allocator does not support multiple heaps. The configuration $\textbf{Base}_{\text{OA}}$ is benchmarks compiled with *vanilla* LLVM but running with our custom allocator.
- $\textbf{Our}_{\text{Bare}}$: Compiled with CONFLLVM, but without any runtime instrumentation. However, all optimizations unsupported by CONFLLVM are disabled, the memories of $\mathcal{T}$ and $\mathcal{U}$ are separated and, hence, stacks are switched in calling $\mathcal{T}$ functions from $\mathcal{U}$.
- $\textbf{Our}_{\text{CFI}}$: Like $\textbf{Our}_{\text{Bare}}$ but additionally with CFI instrumentation, but no memory bounds enforcement.
- $\textbf{Our}_{\text{MPX}}$: Full CONFLLVM, using MPX scheme.
- $\textbf{Our}_{\text{Seg}}$: Full CONFLLVM, using segmentation scheme.

Briefly, the difference between $\textbf{Our}_{\text{CFI}}$ and $\textbf{Our}_{\text{Bare}}$ is the cost of our CFI instrumentation. The difference between $\textbf{Our}_{\text{MPX}}$ (resp. $\textbf{Our}_{\text{Seg}}$) and $\textbf{Our}_{\text{CFI}}$ is the cost of enforcing bounds using MPX (resp. segment registers).

All benchmarks were run on a Microsoft Surface Pro-4 Windows 10 machine with an Intel Core i7-6650U 2.20 GHz 64-bit processor with 2 cores (4 logical cores) and 8 GB RAM. Table 6 shows the results, averaged over ten runs. The numbers in the first column are run times in seconds. The remaining columns report percentage changes, relative to the first column. The standard deviations were all below 3%.

The overhead of CONFLLVM using MPX ($\textbf{Our}_{\text{MPX}}$) is up to 74.03%, while that of CONFLLVM using segmentation ($\textbf{Our}_{\text{Seg}}$) is up to 24.5%.[6] As expected, the overheads are almost consistently significantly lower when using segmentation than when using MPX. Looking further, some of the overhead (up to 10.2%) comes from CFI enforcement ($\textbf{Our}_{\text{CFI}} - \textbf{Our}_{\text{Bare}}$), although the average CFI overhead is 3.62%, competitive with best known techniques [21]. Stack switching and disabled optimizations ($\textbf{Our}_{\text{Bare}}$) account for the remaining overhead. The overhead due to our custom memory allocator ($\textbf{Base}_{\text{OA}}$) is negligible and, in many benchmarks, the custom allocator improves performance.

We further comment on some seemingly odd results. On mcf, the cost of CFI alone ($\textbf{Our}_{\text{CFI}}$, 4.17%) seems to be higher than that of the full MPX-based instrumentation ($\textbf{Our}_{\text{MPX}}$, 4.02%). We verified that this is due to an outlier in the $\textbf{Our}_{\text{CFI}}$ experiment. On hmmer, the overhead of $\textbf{Our}_{\text{Bare}}$ is negative because the optimizations that CONFLLVM disables actually slow it down. Finally, on milc, the overhead of CONFLLVM is negative because this benchmark benefits significantly from the use of our custom memory allocator. Indeed, relative to $\textbf{Base}_{\text{OA}}$, the remaining overheads follow expected trends.

Name	**Base**(s)	$\textbf{Base}_{\text{OA}}$	$\textbf{Our}_{\text{Bare}}$	$\textbf{Our}_{\text{CFI}}$	$\textbf{Our}_{\text{Seg}}$	$\textbf{Our}_{\text{MPX}}$
gcc	272.36	-5.92%	0.17%	3.37%	5.48%	12.32%
gobmk	395.62	0.08%	4.32%	13.66%	20.90%	33.13%
hmmer	128.63	0.16%	-2.54%	-2.50%	1.16%	65.09%
h264ref	365.13	-0.63%	6.96%	17.12%	22.20%	74.03%
lbm	208.80	0.82%	6.78%	7.06%	10.63%	11.84%
bzip2	406.33	-0.47%	3.39%	4.62%	10.19%	40.95%
mcf	280.06	1.32%	2.11%	4.17%	3.69%	4.02%
milc	438.60	-8.81%	-7.61%	-7.17%	-6.40%	-3.22%
libquantum	263.8	2.76%	3.831%	6.25%	17.89%	40.75%
sjeng	424.46	0.01%	12.61%	19.75%	24.5%	48.9%
sphinx3	438.6	0.93%	1.74%	5.26%	9.93%	30.30%

FIGURE 6: SPEC CPU 2006 BENCHMARKS IN DIFFERENT CONFIGURATIONS. PERCENTAGES ARE OVERHEADS RELATIVE TO THE FIRST COLUMN.

B. *Integrity of system libraries*

Next, we should how to use CONFLLVM to isolate a shared system library from an untrusted client program. The goal in this setting is integrity of the library's internal data; this goal is dual to confidentiality. As an example, we wrote a small multithreaded, shared userspace library that offers file `read` and `write` functions. Internally, the library memory-maps open files, and serves and updates file contents through memory copy. The library also provides data integrity by maintaining a Merkle hash tree of the file system's contents, at the granularity of file system blocks (512 bytes).

The obvious security concern is that malicious or buggy applications may clobber the hash tree and nullify the integrity guarantees. To alleviate this concern we compile both the library and its clients using CONFLLVM (i.e. as part of $\mathcal{U}$). *All* data within the client is marked *private*, while the integrity-sensitive datastructures of the library, including the hash tree, are marked *public* (the library can also work with the private data). While marking the library data public and the client data private may sound backwards, note that the goal is here is to protect the integrity, not the confidentiality, of library data. Marking the data this way prevents compromised clients from clobbering library data structures. It also prevents the library from accidentally copying client data into its own data structures. Finally, we also prevent clients from calling internal library functions directly (for this, we use different magic sequences for the client and the library). Client calls to the library functions are mediated through wrapper functions in $\mathcal{T}$, that switch back and forth between the library's public stack and the client's stack.[7]

[5] O2 is the standard optimization level for performance evaluation. Higher levels include "optimizations" that don't always speed up the program.

[6] The overheads of MPX may seem high, especially given that MPX was designed to make memory-bounds checking efficient. However, Oleksenko *et al.*'s recent investigation of MPX has found very similar overheads [39].

[7] An alternative design could be to move the entire library to $\mathcal{T}$, while keeping the application in $\mathcal{U}$. Compared to our design, this design has the disadvantage that it does not prevent bugs in the library from accidentally copying application data into its critical data structures.

Threads	**Base**	**Our**$_{\text{Seg}}$	**Our**$_{\text{MPX}}$
1	16.0	17.5 (9.38%)	18.6 (16.25%)
2	16.4	18.0 (9.76%)	19.1 (16.46%)
4	17.5	19.1 (9.14%)	20.2 (15.43%)
6	26.1	28.7 (9.96%)	30.5 (16.85%)

Figure 7: Total time, in seconds, for reading a 2GB file in parallel, as a function of the number of threads. Numbers in parenthesis are overheads relative to Base.

We experiment with this library on a Windows 10 machine with an Intel i7-6700 CPU (4 cores, 8 hyperthreaded cores) and 32 GB RAM. Our client program creates between 1 and 6 parallel threads, all of which read a 2 GB file concurrently. The file is memory-mapped within the library and cached previously. This gives us a CPU-bound workload. We measure the total time taken to perform the reads in three configurations: **Base**, **Our**$_{\text{Seg}}$ and **Our**$_{\text{MPX}}$. Table 7 shows the total runtime (in seconds) as a function of the number of threads and the configuration, averaged across 5 runs. The standard deviations are negligible ($< 3\%$). Until the number of threads exceeds the number of cores (4), the time and relative overhead of our scheme remain nearly constant, establishing linear scaling with the number of threads. The actual overhead of **Our**$_{\text{Seg}}$ is below 10% and that of **Our**$_{\text{MPX}}$ is below 17% in this experiment.

C. OpenLDAP

In order to show that ConfLLVM can scale to large applications, we use it to compile the OpenLDAP project [40]. OpenLDAP is an open source implementation of the Lightweight Directory Access Protocol (LDAP) proposed by the IETF RFC 4511 [46]. LDAP provides a standardized way of organizing information in an application-defined hierarchical structure, as well as accessing the information over a network. We use OpenLDAP version 2.4.45. This version has 300,000 lines of C code, across 728 source files. We configure OpenLDAP as a multi-threaded server (the default) with a memory-mapped backing store (also the default), and simple username/password authentication.

By default, OpenLDAP stores two types of passwords: 1) A password for each user in the database; authenticating with this password provides access to specific parts of the database, and 2) A root password that gives access to the *entire* backing store. We modified the source code to protect both the types by changing 100 lines of code, adding 52 lines of new $\mathcal{T}$ code, and moving 9 lines of existing code into $\mathcal{T}$. In total, these constitute about 0.5% of the existing code base.

We briefly describe our code changes and additions. In OpenLDAP, the root password is stored in a configuration file that is read during initialization. To protect the root password, we moved the root password to a different location and added a $\mathcal{T}$ function to read the password from this location into a private buffer. Subsequently, ConfLLVM guarantees that this password cannot leak to the public region. To protect user passwords, we changed OpenLDAP to encrypt these passwords when they are written to the backing store and to decrypt them when they are read back. The functions to encrypt and decrypt are in $\mathcal{T}$. Importantly, the decryption function returns the decrypted password in a private buffer. ConfLLVM then prevents this password from leaking.

We perform two throughput experiments on OpenLDAP compiled with ConfLLVM. We use MPX for bounds checks, as opposed to segmentation, since we know from the SPEC CPU benchmarks that MPX is worse for ConfLLVM. We use the same machine as for the SPEC CPU benchmarks to host an OpenLDAP server configured to run 6 concurrent threads. The server is pre-populated with 10,000 random directory entries and its caches are warmed ahead of time.

In the first experiment, 80 concurrent clients, running on a separate machine connected to the server over a 100Mbps direct Ethernet link, issue concurrent requests for directory entries that do not exist. Across three trials, the server handles on average 26,254 and 22,908 requests per second in the baseline (**Base**) and ConfLLVM using MPX (**Our**$_{\text{MPX}}$). This corresponds to a throughput degradation of 12.74%, which is moderate. The server CPU remains nearly saturated throughout the experiment. The standard deviations are very small (1.7% and 0.2% in **Base** and **Our**$_{\text{MPX}}$, respectively).

Our second experiment is identical to the first, except that 60 concurrent clients issue small requests for entries that do exist on the server. Now, the baseline and ConfLLVM using MPX handle 29,698 and 26,895 queries per second, respectively. This is a throughput degradation of 9.44%. The standard deviations are small (less than 0.2% in both cases).

The reason for the difference in overheads in these two experiments is that OpenLDAP does less work in $\mathcal{U}$ looking for directory entries that exist than it does looking for directory entries that don't exist. We expect that the overheads can be reduced even more by using segmentation in place of MPX.

D. Web server: NGINX

As a second large use-case, we use ConfLLVM to compile NGINX, the most popular web server among high-traffic websites [9]. NGINX has a logging module that logs time stamps, processing times, etc. for each request along with request metadata such as the client address and the request URI. An obvious confidentiality concern is that sensitive content from files being served may leak into the logs due to bugs. We use ConfLLVM to prevent such leaks.

We annotate NGINX's codebase to place OpenSSL in $\mathcal{T}$, and the rest of NGINX, including all its request parsing, processing, serving, and logging code in $\mathcal{U}$. All the code in $\mathcal{U}$ is compiled with ConfLLVM (total 124,001 LoC). Within $\mathcal{U}$, we mark everything as `private`, except for the buffers in the logging module that are marked as `public`. To log request URIs, which are actually private, we add a new `encrypt_log` function to $\mathcal{T}$ that $\mathcal{U}$ invokes to encrypt the request URI before adding it to the log. This function encrypts using a key that is isolated in $\mathcal{T}$'s own region. The key is pre-shared with the administrator who is authorized to read the logs. The encrypted result of `encrypt_log` is placed in a public buffer. The standard `SSL_recv` function in $\mathcal{T}$ decrypts the incoming payload with the session key, and provides it to $\mathcal{U}$ in a private buffer.

File size	**Base**	**Our**$_{\text{1Mem}}$	**Our**$_{\text{Bare}}$	**Our**$_{\text{CFI}}$	**Our**$_{\text{MPX-Sep}}$	**Our**$_{\text{MPX}}$
0	72,917	-3.19%	6.56%	14.30%	18.86%	18.92%
1 KB	65,891	-2.34%	6.37%	14.17%	15.80%	20.27%
10 KB	50,959	4.2%	10.15%	19.40%	18.30%	29.32%
20 KB	38,158	2.54%	5.20%	8.39%	11.40%	12.18%
40 KB	24,938	0.3%	-2.68%	-0.33%	1.61%	3.25%

FIGURE 8: NGINX THROUGHPUT IN REQ/S IN DIFFERENT CONFIGURATIONS FOR DIFFERENT REQUEST SIZES.

```
size_t SSL_recv(SSL *connection, private void *buffer,
                size_t length);
```

In total, we added or modified 160 LoC in NGINX (0.13% of NGINX's codebase) and added 138 LoC to $\mathcal{T}$.

Our goal is to measure the overhead of CONFLLVM on the sustained throughput of NGINX. We run our experiments in 6 configurations: **Base**, **Our**$_{\text{1Mem}}$, **Our**$_{\text{Bare}}$, **Our**$_{\text{CFI}}$, and **Our**$_{\text{MPX-Sep}}$, **Our**$_{\text{MPX}}$. Of these, **Base**, **Our**$_{\text{Bare}}$, **Our**$_{\text{CFI}}$, and **Our**$_{\text{MPX}}$ are the same as those in Section VIII-A. **Our**$_{\text{1Mem}}$ is like **Our**$_{\text{Bare}}$ (compiled with CONFLLVM without any instrumentation), but also does not separate memories for $\mathcal{T}$ and $\mathcal{U}$. **Our**$_{\text{MPX-Sep}}$ includes all instrumentation, but does not separate stacks for private and public data. Briefly, the difference between **Our**$_{\text{Bare}}$ and **Our**$_{\text{1Mem}}$ is the overhead of separating $\mathcal{T}$'s memory from $\mathcal{U}$'s and switching stacks on every call to $\mathcal{T}$, while the difference between **Our**$_{\text{MPX}}$ and **Our**$_{\text{MPX-Sep}}$ is the overhead of increased cache pressure from having separate stacks for private and public data.

We host an NGINX server on an Intel Core i7-6700 3.40GHz 64-bit processor with 4 cores (8 logical cores), 32 GB RAM, and a 10 Gbps Ethernet card, running Ubuntu v16.04.1 with Linux kernel version 4.13.0-38-generic. Hyperthreading was disabled. We use NGINX version 1.13.12, configured to run a single worker process pinned to a single core. We connect to the server from two client machines using the `wrk2` tool [12], simulating a total of 32 concurrent clients. Each client makes 10,000 random requests one after the other from a corpus of 1,000 files of the same size (we vary the file size across experiments). The files are served from a RAM disk. This saturates the CPU core hosting NGINX in all setups. The numbers reported below are averages of 10 runs. The standard deviations are all below 0.3%, except in **Base** for file sizes 0 KB and 1 KB, where the standard deviations are below 2.2%.

Figure 8 shows the throughputs in the six configurations for file sizes ranging from 0 to 40 KB. The throughput for **Base** is reported in requests/s and the remaining columns are percentage changes relative to **Base**. For file sizes beyond 40 KB, the 10 Gbps network card saturates in the base line before the CPU, and the excess CPU cycles absorb our overheads.

Overall, CONFLLVM's overhead on throughput ranges from 3.25% to 29.32%. The overhead is not monotonic in file size or base line throughput, indicating that there are compensating effects at play. For large file sizes, the relative amount of time spent outside $\mathcal{U}$, e.g., in the kernel in copying data, is substantial. Since code outside $\mathcal{U}$ is not subject to our instrumentation, our overhead falls for large file sizes (>10 KB here) and eventually tends to zero. The initial increase in overhead up to file size 10 KB comes mostly from the increased cache pressure due to the separation of stacks for public and private data (the difference **Our**$_{\text{MPX}}-$**Our**$_{\text{MPX-Sep}}$). This is unsurprising: As the file size increases, so does the cache footprint of $\mathcal{U}$. In contrast, the overheads due to CFI (difference **Our**$_{\text{CFI}}-$**Our**$_{\text{Bare}}$) and the separation of the memories of $\mathcal{T}$ and $\mathcal{U}$ (difference **Our**$_{\text{Bare}}-$**Our**$_{\text{1Mem}}$) are relatively constant for small file sizes.

E. *Vulnerability-injection experiments*

To test that CONFLLVM stops data extraction vulnerabilities from being exploited, we hand-crafted vulnerabilities in three applications. First, we introduced a buffer-bounds vulnerability in the popular Mongoose web server [8] in the code path for serving an unencrypted file (over http). The vulnerability transmits any amount of stale data from the stack, unencrypted. We wrote a client that exploits the vulnerability by first requesting a private file, which causes some contents of the private file to be written to the stack in clear text, and then requesting a public file with the exploit. This causes stale private data from the first request to be leaked. Using CONFLLVM on Mongoose with proper annotations stops the exploit since CONFLLVM separates the private and public stacks. The contents of the private file are written to the private stack but the vulnerability reads from the public stack.

Second, we modified Minizip [7], a file compression tool, to explicitly leak the file encryption password to a log file. CONFLLVM's type inference detects this leak once we annonate the password as `private`. To make it harder for CONFLLVM, we added several pointer type casts on the password, which make it impossible to detect the leak statically. But then, the dynamic checks inserted by CONFLLVM prevent the leak.

Third, we wrote a simple function with a format string vulnerability in its use of `printf`. `printf` is a vararg function, whose first argument, the format string, determines how many subsequent arguments the function tries to print. If the format string has more directives than the number of arguments, potentially due to adversary provided input, `printf` ends up reading other data from either the argument registers or the stack. If any of this data is private, it results in a data leak. CONFLLVM prevents this vulnerability from being exploited if we include `printf`'s code in $\mathcal{U}$: since `printf` tries to read all arguments into buffers marked public, the bounds enforcement of CONFLLVM prevents it from reading any private data.

IX. DISCUSSION AND FUTURE WORK

Our scheme implements information-flow control for a two-level ({private, public}) lattice. While this is sufficient for our goal of preventing leaks of sensitive data, it cannot be used for finer-grained control, e.g. to prevent one user's data from flowing to a different user in a multi-tenant application. Our design leverages the two-level restriction in the segment-register scheme of Section III that uses the two available segment registers. The MPX scheme is more general, but we leave the support for more complex lattices as future work.

We also do not have label polymorphism, although that can potentially be implemented using C++ templates.

In our scheme, $\mathcal{T}$ is trusted and therefore must be implemented with care. The scheme guarantees that $\mathcal{U}$ cannot access $\mathcal{T}$'s memory or jump to arbitrary points in $\mathcal{T}$; hence the only remaining attack surface for $\mathcal{T}$ is the API that it exposes to $\mathcal{U}$. $\mathcal{T}$ must ensure that stringing together a sequence of these API calls cannot cause leaks. We recommend the following (standard) strategies. First, $\mathcal{T}$ should be kept small, mostly containing generic, application-independent functionality, e.g. communication interface, cryptographic routines, and optionally a small number of libc routines (mainly for performance reasons), moving the rest of the code to $\mathcal{U}$. This helps ensure that $\mathcal{T}$ is reused for multiple applications and can be subject to careful audit/verification. Further, declassification routines in $\mathcal{T}$ must provide guarded access to $\mathcal{U}$. For instance, $\mathcal{T}$ can disallow an arbitrary number of calls to a password checking routine (e.g. raising an alert and halting the application on some consecutive failed attempts).

We rely on the absence of magic sequence in the $\mathcal{T}$ binary to prevent $\mathcal{U}$ from jumping inside (uninstrumented) $\mathcal{T}$. We ensure this by selecting the magic string when the entire code of $\mathcal{U}$ and $\mathcal{T}$ is available. This still leaves the problem of dynamic loading. While dynamic loading in $\mathcal{U}$ is disallowed, any dynamic loading in $\mathcal{T}$ must ensure that the loaded library does not contain the magic sequence. By generating the (59-bits) magic sequence at random, the chances of it appearing in the loaded library is minimal. It can also be enforced if one can over-approximate the set of libraries that may be loaded at runtime. Another possible defense is to guard indirect control transfers to remain inside $\mathcal{U}$'s own code.

ConfLLVM supports callbacks from $\mathcal{T}$ to $\mathcal{U}$ with the help of *trusted* wrappers in $\mathcal{U}$ that return to a fixed location in $\mathcal{T}$, where $\mathcal{T}$ can restore its stack and start execution from where it left off (or fail if $\mathcal{T}$ never called into $\mathcal{U}$). We leave this support in ConfVerify and our formal model as future work.

X. Related Work

Our work bears similarities to Sinha et al. [50] who proposed a design methodology for programming secure enclaves (e.g., those that use Intel SGX instructions for memory isolation). The code inside an enclave is divided into $\mathcal{U}$ and $\mathcal{L}$. $\mathcal{U}$'s code is compiled via a special instrumenting compiler [19] while $\mathcal{L}$'s code is trusted and may be compiled using any compiler. This is similar in principle to our $\mathcal{U}$-$\mathcal{T}$ division. However, there are several differences. First, even the goals are different: their scheme does not track taints; it only ensures that all unencrypted I/O done by $\mathcal{U}$ goes through $\mathcal{L}$, which encrypts all outgoing data uniformly. Thus, the application cannot carry out plain-text communication without losing the security guarantee. Second, their implementation does not support multi-threading (it relies on page protection to isolate $\mathcal{L}$ from $\mathcal{U}$). Third, they maintains a *bitmap* of writeable memory locations for enforcing CFI resulting in time and memory overheads. Our CFI is taint-aware and without these overheads. Finally, their verifier does not scale even for SPEC benchmarks, whereas our verifier is much faster and scales to all binaries that we have tried.

In an effort parallel to ours, Carr *et al.* [18] present DataShield, whose goal, like ConfLLVM's, is information flow control in low-level code. However, there are several differences between DataShield and our work. First and foremost, DataShield itself only prevents *non-control* data flow attacks in which data is leaked or corrupted without relying on a control flow hijack. A separate CFI solution is needed to prevent leaks of information in the face of control flow hijacks. In constrast, our technique prevents information leaks even in the presence of control flow hijacks. One of the key insights of our work is that (standard) CFI is neither necessary nor sufficient to prevent information flow violations due to control flow hijacks. We integrate a novel taint-aware CFI into ConfLLVM for exactly this purpose. Second, DataShield places blind trust in its compiler. In constrast, in our work, the verifier ConfVerify eliminates the need to trust the compiler ConfLLVM. Third, DataShield enforces memory safety at object-granularity on sensitive objects. This allows DataShield to enforce *integrity* for data invariants, which is mostly outside the scope of our work. However, as we show in Section VIII-B, our work can be used to prevent untrusted data from flowing into sensitive locations, which is a different form of integrity.

Region-based memory partitioning has been explored before in the context of safe and efficient memory management [52], [25], but not for information flow. In ConfLLVM, regions obviate the use of dynamic taint tracking [48], [34], [43]. TaintCheck [38] first proposed the idea of dynamic taint tracking, and forms the basis for Valgrind [37]. DECAF [28] is a whole system binary analysis framework including a taint-tracking mechanism. However, such dynamic taint trackers incur heavy performance overhead. E.g., DECAF has an overhead of 600%. Similarly, TaintCheck can impose a >37x performance hit for CPU-bound applications. Suh *et al.* [51] report less than 1% overheads for their dynamic information flow tracking scheme, but they rely on custom hardware.

Static analyses for security [16], [17], [26], [45] use source code to prove correctness criteria such as safe downcasts in C++, or the correct use of variadic arguments. When proofs cannot be constructed, runtime checks are inserted to enforce relevant policy at runtime. This is similar to our use of runtime checks as it is not always possible to determine statically whether a pointer belongs to the private or public partition.

Memory-safety techniques for C such as CCured [36] and SoftBound [35] do not provide confidentiality in all cases and already have overheads higher than those of ConfLLVM (see [18, Section 2.2] for a summary of overheads of existing memory-safety techniques). Techniques such as control flow integrity (CFI) [13] and code-pointer integrity (CPI) [30] prevent control flow hijacks but not all data leaks. While our new taint-aware CFI is an integral component of our enforcement, our goal of preventing data leaks goes beyond CFI and CPI. Our CFI mechanism is similar to Abadi et al. [13] and Zeng et al. [54] in its use of magic sequences, but our magic sequences are taint-aware.

References

[1] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[2] Chrome owned by exploits in hacker contests, but google's $1m purse still safe. https://www.wired.com/2012/03/pwnium-and-pwn2own/.

[3] Clang: A C language family frontend for LLVM. http://clang.llvm.org.

[4] Cve-2012-0769, the case of the perfect info leak. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[5] The heartbleed bug. http://heartbleed.com/.

[6] Intel mpx explained. https://intel-mpx.github.io/.

[7] Minizip. https://github.com/nmoinvaz/minizip.

[8] Mongoose. https://github.com/cesanta/mongoose.

[9] NGINX web server. https://www.nginx.com/.

[10] Smashing the stack for fun and profit. insecure.org/stf/smashstack.html.

[11] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[12] wrk2: A constant throughput, correct latency recording variant of wrk . https://github.com/giltene/wrk2.

[13] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[14] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[15] Ken Biba. Integrity considerations for secure computer systems. page 68, 04 1977.

[16] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. Venerable variadic vulnerabilities vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[17] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. *SIGPLAN Not.*, 51(4):143–157, March 2016.

[18] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 193–204, New York, NY, USA, 2017. ACM.

[19] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[20] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[24] Jeffrey S. Foster, Manuel F&quot;ahndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, Atlanta, Georgia, May 1999.

[25] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[26] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. ACM.

[27] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

[28] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, Feb 2017.

[29] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[30] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[31] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[34] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 308–319, New York, NY, USA, 2016. ACM.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 245–258, 2009.

[36] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[37] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[38] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[39] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017.

[40] OpenLDAP Project. Openldap. http://www.openldap.org/.

[41] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[43] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 15–30, March 2016.

[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[46] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511, June 2006.

[47] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th*

ACM Conference on Computer and Communications Security, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[48] Zhiyong Shan. Suspicious-taint-based access control for protecting OS from network attacks. *CoRR*, abs/1609.00100, 2016.

[49] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[50] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 665–681, 2016.

[51] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[52] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997.

[53] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[54] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 29–40. ACM, 2011.

APPENDIX

Figures 9 and 10 present the operational semantic of commands in our modeling language and the complete list of rules in the type system. For the operation rules we use $e \Downarrow n$ to mean evaluation of an expression to its corresponding (concrete) value. Moreover, we extend the configuration s of G with an additional component ν that represents the memory fragment available only to the trusted code and its protected against untrusted accesses, i.e., $\langle \nu, \mu, \rho, [\sigma_{\mathbf{H}} : \sigma_{\mathbf{L}}], pc \rangle$. Further, we introduce the record $\mathcal{F} = \{ f_i \mapsto \langle n_i, M\_call_i \rangle \mid f_i \in \mathcal{P} \}$ and extend the typing judgment with this record $\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma'$. $\mathcal{F}$ keeps for each function its starting node in the CFG and the magic sequence associated with that function.

We also formally define call- and return-sites magic sequences based on the CFG structure. Let $Cons$ be the bit-concatenation function, then: (i) for the function entry node v, $M\_call = \underset{i=0\ldots3}{Cons}(v.\Gamma(reg_i))$, and (ii) for a given return address adr and the node v such that $v \in pred(G, adr)$, $M\_ret = v.\Gamma'(reg_0)$. We now turn to explain the meaning of rules in Figure 10. Here we try to explain rules which are less intuitive.

a) **call***:* For the **call** statement we check that the expected taints of the arguments, as encoded in the magic sequence at the callee, matches the taints of the argument registers at the callsite. It is also worth noting that at runtime all return addresses are stored in the stack allocated for the low-security context.

b) **icall***:* For the indirect calls CONFVERIFY confirms that the function pointer is low and that there is a check for the magic sequence at the target site and its taint bits match the inferred taints for registers. Note that since the check $\textbf{assert}(\ell_{[1-4]} \sqsubseteq M\_call$ **and** $e_f \in G$ **and** $(e_f \mapsto M\_call) \in$

$$\frac{C = \textbf{ldr}(reg, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \in (Dom(\mu_{\mathbf{L}}) \cup Dom(\mu_{\mathbf{H}}))}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho[reg \mapsto \mu(v)], [\sigma], pc + 1 \rangle}$$

$$\frac{C = \textbf{ldr}(reg, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \notin (Dom(\mu_{\mathbf{L}}) \cup Dom(\mu_{\mathbf{H}}))}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \lightning}$$

$$\frac{C = \textbf{str}(reg, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \in (Dom(\mu_{\mathbf{L}}) \cup Dom(\mu_{\mathbf{H}}))}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu[v \mapsto \rho(reg)], \rho, [\sigma], pc + 1 \rangle}$$

$$\frac{C = \textbf{str}(reg, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \notin (Dom(\mu_{\mathbf{L}}) \cup Dom(\mu_{\mathbf{H}}))}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \lightning}$$

$$\frac{C = \textbf{call}_{\mathcal{U}} \;\; f_u\,(e_1, \ldots, e_4) \quad (f_u \mapsto \langle pc_f, - \rangle) \in \mathcal{F}}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [(pc+1); \sigma_{\mathbf{L}} : \sigma_{\mathbf{H}}], pc_f \rangle}$$

$$\frac{C = \textbf{call}_{\mathcal{T}} \;\; f_{\mathcal{T}}\,(e_1, \ldots, e_4) \quad (f_{\mathcal{T}} \mapsto \langle pc_f, - \rangle) \in \mathcal{F}}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \hookrightarrow_{f_{\mathcal{T}}} \langle \nu', \mu', \rho', [(pc+1); \sigma'_{\mathbf{L}} : \sigma'_{\mathbf{H}}], pc_f \rangle}$$

$$\frac{C = \textbf{icall}\, e_f\,(e_1, \ldots, e_4) \quad \mu, \rho \vdash e_f \Downarrow pc_f \quad pc_f \in G}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [(pc+1); \sigma_{\mathbf{L}} : \sigma_{\mathbf{H}}], pc_f \rangle}$$

$$\frac{C = \textbf{icall}\, e_f\,(e_1, \ldots, e_4) \quad \mu, \rho \vdash e_f \Downarrow pc_f \quad pc_f \notin G}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \lightning}$$

$$\frac{C = \textbf{ret} \quad adr \in G}{\langle \nu, \mu, \rho, [adr; \sigma_{\mathbf{L}} : \sigma_{\mathbf{H}}], pc \rangle \rightarrow \langle \nu, \mu, \rho, [\sigma_{\mathbf{L}} : \sigma_{\mathbf{H}}], adr] \rangle}$$

$$\frac{C = \textbf{ret} \quad adr \notin G}{\langle \nu, \mu, \rho, [adr; \sigma_{\mathbf{L}} : \sigma_{\mathbf{H}}], pc \rangle \rightarrow \lightning}$$

$$\frac{C = \textbf{goto}\,(e) \quad \mu, \rho \vdash e \Downarrow pc}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [\sigma], pc] \rangle}$$

$$\frac{C = \textbf{ifthenelse}\,(e, \textbf{goto}\,(e_1)\,, \textbf{goto}\,(e_2)) \quad \mu, \rho \vdash e \Downarrow T \quad \mu, \rho \vdash e_1 \Downarrow n_1}{\langle \nu, \mu, , \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [\sigma], n_1 \rangle}$$

$$\frac{C = \textbf{ifthenelse}\,(e, \textbf{goto}\,(e_1)\,, \textbf{goto}\,(e_2)) \quad \mu, \rho \vdash e \Downarrow F \quad \mu, \rho \vdash e_2 \Downarrow n_2}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [\sigma], n_2 \rangle}$$

$$\frac{C = \textbf{assert}(e) \quad \mu, \rho \vdash e \Downarrow T}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle \nu, \mu, \rho, [\sigma], pc + 1 \rangle}$$

$$\frac{C = \textbf{assert}(e) \quad \mu, \rho \vdash e \Downarrow F}{\langle \nu, \mu, \rho, [\sigma], pc \rangle \rightarrow \bot}$$

FIGURE 9: OPERATIONAL SEMANTICS RULES.

$$\frac{\begin{array}{c}C = \textbf{ldr}(reg, e)\\ \forall\ v \in pred(G, pc).\ v.C = \textbf{assert}(e \in Dom(\mu_{\ell_e}))\end{array}}{\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma[reg \mapsto \ell_e]}$$

$$\frac{\begin{array}{c}C = \textbf{str}(reg, e) \quad \Gamma \vdash reg : \ell_r \quad \ell_r \sqsubseteq \ell_e\\ \forall v \in pred(G, pc).\ v.C = \textbf{assert}(e \in Dom(\mu_{\ell_e}))\end{array}}{\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma}$$

$$\frac{\begin{array}{c}C = \textbf{call}_{\{\mathcal{U}|\mathcal{T}\}}\ f\,(e_1, \ldots, e_4) \quad (f \mapsto \langle -, M\_call\rangle) \in \mathcal{F}\\ \Gamma \vdash e_i : \ell_i \quad \text{for} \quad i = 1 \ldots 4 \qquad \ell_{[1-4]} \sqsubseteq M\_call\end{array}}{\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma[rcaller \mapsto \mathbf{H}, rcallee \mapsto \mathbf{L}]}$$

$$\frac{\begin{array}{c}C = \textbf{icall}\, e_f(e_1, \ldots, e_4) \quad \Gamma \vdash e_f : \ell_f \sqsubseteq \mathbf{L} \quad \Gamma \vdash e_i : \ell_i\ \textbf{for}\ i = 1 \ldots 4\\ \forall v \in pred(G, pc).\ v.C = \textbf{assert}\left(\begin{array}{ll}\ell_{[1-4]} \sqsubseteq M\_call & \textbf{and}\\ e_f \in G & \textbf{and}\\ (e_f \mapsto \langle -, M\_call\rangle) \in \mathcal{F}) & \end{array}\right)\end{array}}{\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma[rcaller \mapsto \mathbf{H}, rcallee \mapsto \mathbf{L}]}$$

$$\frac{\begin{array}{c}C = \textbf{ret} \quad \Gamma \vdash rcallee \sqsubseteq \mathbf{L} \quad \Gamma \vdash reg_0 : \ell\\ \forall v \in pred(G, pc).\ v.C = \textbf{assert}(\forall v' \in pred(G, top(\sigma_{\mathbf{L}})).\ \ell \sqsubseteq v'.\Gamma'(reg_0))\end{array}}{\mathcal{F}, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

$$\frac{C = \textbf{goto}\,(e) \quad \Gamma \vdash e : \ell_e \sqsubseteq \mathbf{L}}{\mathcal{F}, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

$$\frac{C = \textbf{ifthenelse}\,(e, \textbf{goto}\,(e_1), \textbf{goto}\,(e_2)) \quad \Gamma \vdash e : \ell_e \sqsubseteq \mathbf{L}}{\mathcal{F}, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

FIGURE 10: COMPLETE LIST OF TYPE RULES. C IS A COMMAND FROM THE CFG NODE POINTED TO BY pc AND $rcallee$ AND $rcaller$ ARE CALLEE- AND CALLER-SAVE REGISTERS.

$\mathcal{F}$) is a runtime condition, the pointer e_f will be evaluated to the corresponding function name at the execution time and we can retrieve the magic string directly from $\mathcal{F}$. The condition $e_f \in G$ ensures that the target of the call statement is a valid node in the CFG of the program.

c) **ret***:* Similar to indirect function call, for the **ret** command we check that there is a check for the magic sequence at the target site and that its taint bits match the inferred taints for registers. Additionally, CONFVERIFY confirms that the return address on the stack has a magic signature that has a taint bit of inferred return type ℓ for the function. In this rule we use the function top with the standard meaning to manipulate the stack content. Again since **assert**$(\forall v' \in pred(G, top(\sigma_{\mathbf{L}})).\ \ell \sqsubseteq v'.\Gamma'(reg_0))$ is checked at runtime, we will have access to the stack.

d) **ifthenelse***:* For this rule we require that the security level of conditional expression is not **H**. Checks on the **goto** and **ifthenelse** guarantee that the program flow is secret independent.