What you get is what you C: Controlling side effects in mainstream C compilers

Laurent Simon
Samsung Research America
University of Cambridge
lmrs2@cl.cam.ac.uk

David Chisnall
University of Cambridge
David.Chisnall@cl.cam.ac.uk

Ross Anderson
University of Cambridge
Ross.Anderson@cl.cam.ac.uk

Abstract

Security engineers have been fighting with C compilers for years. A careful programmer would test for null pointer dereferencing or division by zero; but the compiler would fail to understand, and optimize the test away. Modern compilers now have dedicated options to mitigate this.

But when a programmer tries to control side effects of code, such as to make a cryptographic algorithm execute in constant time, the problem remains. Programmers devise complex tricks to obscure their intentions, but compiler writers find ever smarter ways to optimize code. A compiler upgrade can suddenly and without warning open a timing channel in previously secure code. This arms race is pointless and has to stop.

We argue that we must stop fighting the compiler, and instead make it our ally. As a starting point, we analyze the ways in which compiler optimization breaks implicit properties of crypto code; and add guarantees for two of these properties in Clang/LLVM.

Our work explores what is actually involved in controlling side effects on modern CPUs with a standard toolchain. Similar techniques can and should be applied to other security properties; achieving intentions by compiler commands or annotations makes them explicit, so we can reason about them. It is already understood that explicitness is essential for cryptographic protocol security and for compiler performance; it is essential for language security too. We therefore argue that this should be only the first step in a sustained engineering effort.

1. Introduction

The C language has been used for over 40 years for many of the most critical software components, from operating system kernels to cryptographic libraries. Yet writing reliable, stable, and secure C code remains challenging. Defects can be introduced not just by the programmer directly, but also by the compiler if it makes different assumptions.

The C standard [1] is explicit about *Unspecified Behavior*, *Implementation-defined Behavior*, and *Undefined Behavior (UB)*, which have generated a large body of work, and which a programmer can now control through compiler flags and options (Section 2 on the following page).

However, programmers sometimes have goals or implicit assumptions that they cannot express to compilers, and which lead to subtle failures. Implicit assumptions try to control side effects of source code; but the C standard is concerned only with effects that are visible within the C abstract machine, not about how those effects are generated. Providing explicit control for implicit assumptions is challenging.

There is a long list of implicit properties that would benefit from explicit controls (Section 3 on page 5). For example, cryptographic code has to run in constant time, to forestall timing attacks [2–7]. Unfortunately, there is no way to express this in C, so cryptographers resort to coding tricks instead – which a compiler can frustrate by optimizing away code that seems to do no "useful" work (Section 2 on the following page). And while a clever coder may outwit today's compiler, tomorrow's can quietly open the code to attack.

We already have notable extensions for optimizing performance including the `restrict` keyword to inform alias analysis, explicit alignment and padding attributes (`__attribute__ ((aligned (N)))` and `__attribute__ ((packed))`), and gcc's vector attributes to enable SIMD instructions. Just as better communication between programmers and compilers has led to faster software, better communication with security engineers should lead to better security.

We argue that we must stop fighting the compiler, and instead we must make it our ally. We need a sustained engineering effort to provide explicit compiler support for the implicit side effects of code. Modern compiler IRs differ, but the core requirements for implementing such features are present in gcc, Open64, and so on.

As examples, we implement constant-time selection and register/stack erasure in Clang/LLVM (Section 4 on page 6). We demonstrate that given the right levels of abstraction, programmers can express implicit assumptions with improvements in security, speed and code readability.

In this work, we contribute the following:

- We present the problem of *implicit invariants*: these are increasingly important in security engineering, yet remain a fundamental problem in the toolchain.
- We give examples of implicit invariants that would benefit from explicit controls, with concrete examples from the realm of cryptography.
- We treat the examples of constant-time selection and secret erasure in detail. They are achieved unreliably through implicit, ad-hoc mechanisms today. By adding support directly into the Clang/LLVM framework, we

demonstrate the benefit of making implicit invariants explicit in C programs.
- We make our code available at [8, 9].

2. Compiler vs. Developer

2.1. Explicit Invariants

C was designed to be an efficient low-level programming language, to be portable across architectures, and to enable advanced compiler optimizations. To achieve this goal, it defines an abstract target machine, close enough to real machines to allow compilers to generate fast code, yet far enough to be general. In the process, it leaves certain properties "undefined"; their behavior may vary, although they should be consistent across one implementation.

Unspecified Behavior (USB) is where the standard "provides two or more possibilities and imposes no further requirements on which is chosen in any instance. An example of unspecified behavior is the order in which the arguments to a function are evaluated" [1]. A programmer should avoid using such code as it may behave differently on different platforms, or even when compiled with different toolchains on the same platform.

Implementation-defined Behavior (IDB) deals with "unspecified behavior where each implementation documents how the choice is made" [1]. Examples include the propagation of the high-order bit when a signed integer is shifted right, the size of types, the result of a zero-length `malloc()` (NULL or a unique pointer). Certain such behaviors, such as struct layouts, are defined by the ABI and so are stable over a long time (but not between platforms) whereas others may change when the compiler is updated. gcc provides details in its manual [10].

Undefined Behavior (UB) arises "upon use of a non-portable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements" [1]. UBs are for behaviors difficult to statically prove and expensive to dynamically check: e.g. pointers are not null when dereferenced (branch on every load/store is expensive); memory accesses are all in-bounds (fat pointers are expensive, as are the checks); and memory is not accessed after it is `free()`'d (garbage collection is expensive). What happens to UBs "ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)."

More importantly, the results need not be consistent across the entire code, and this enables advanced compiler optimizations on a case-by-case basis. The speedup comes at the expense of bugs because the results are inconsistent across the code and often counter-intuitive for programmers [11]. Programmers tend to use UBs unintentionally, and this leads to subtle security problems in practice. For example, divisions by zero are considered UB, so a compiler can assume the divisor is always non-zero (Chapter 6.5.5). This means code that checks the value of the divisor prior to a division can be removed by a compiler – we refer the reader to Wang *et al.* [12] for details.

2.1.1. Explicit Control Options. USBs, IDBs, and UBs are explicit in the C standard, and have generated a large body of work [11–17]. As a result, programmers can often control a compiler's explicit assumptions by means of compiler flags and options. We list some of them next.

- Signed integer overflows do not wrap around on all architectures. So the C standard leaves signed integer overflows as UB. Because this can be counter-intuitive, the `-fwrapv` option has been added to gcc to assume signed integer wraparound for addition, subtraction, and multiplication. This is used to compile the Linux kernel.
- Pointer arithmetic never wraps around according to the C standard: so the `-fno-strict-overflow` option was added to gcc to assume pointer arithmetic wraparound. This is also necessary to compile the Linux kernel [18].
- Null pointer dereference is also an explicit UB, so a compiler can remove a program's check: this led to a critical vulnerability in SELinux [19]. An option called `-fno-delete-null-pointer-checks` was created to assume unsafe null pointer dereferences. This is also used to compile the Linux kernel.
- Pointer casts are often used by programmers to reinterpret a given object with a different type: this is called *type-punning*. But the C standard has strict, explicit rules about this: it says that pointers of different types cannot alias. In practice, this can lead to the reordering of read/writes and cause issues [12]. An option called `-fno-strict-aliasing` was created to remove this explicit assumption. This is used to compile the Linux kernel.
- Uninitialized read is considered UB, so a compiler may remove code that computes data from uninitialized memory. But uninitialized memory could be used as an additional source of entropy, albeit weak, to a pseudorandom number generator (PRNG). Removing the code would therefore disable the PRNG altogether. Code removal caused by an uninitialized read was reported in FreeBSD's PRNG [12]. In the Debian OpenSSH fiasco, someone explicitly removed an uninitialized read, dramatically reducing the amount of entropy used in ssh key generation for several years [20].

2.2. Implicit Invariants

As well as these explicit assumptions, there are many implicit invariants or assumptions with which programmers want to endow their code, yet which are not captured by the C standard and current compiler options. Unlike explicit invariants, implicit invariants attempt to control side effects of the code (e.g. timing or memory access pattern). But because side effects were purposely left out of the C standard, developers cannot express these invariants today. Yet, in the realm of cryptography, such invariants are crucial; and cryptographers have been struggling with them for years. Controlling side effects of code is paramount to cryptosystems. For example, it has been known for over twenty years that the time of execution must be indistinguishable

```
uint32_t select_u32(bool b, uint32_t x, uint32_t
    y)
{
  return b ? x : y;
}
```

Listing 1: Naive selection of x or y.

for any key used by a cryptographic algorithm. Failure to maintain time indistuishability has led to key or plaintext recovery time and time again. This may be exploited not just by malicious code running on the same machine as the sensitive code, whether in smartphones [21] or virtualized cloud environments [22–26]; but also sometimes remotely, by protocol counterparties or by wiretappers [27–29]. Yet it is still not possible to control such side effects in modern C compilers today. So it is particularly difficult to control them at the source code level, as we describe next.

2.2.1. Constant-Time Selection. Something as simple as selecting between two variables `x` and `y`, based on a secret selection bit `b` in constant time, is rigged with pitfalls. Naively, the selection function could be written as in Listing 1. But we risk the generated code containing a jump. In fact, if we compile this code for x86 with `gcc` and `Clang` with options `-m32 -march=i386`, the generated code contains a jump regardless of the optimization used. Because of branch prediction, pipeline stalls and/or attacker-controlled cache evictions, the execution time may depend on whether `x` or `y` is returned, leaking the secret bit `b`: this is called a timing side channel vulnerability. Cryptographers must therefore come up with ingenious ways to get a compiler to respect implicit invariants. The usual idea is to write obfuscated code to outwit the compiler, hoping it will not spot the trick and optimize it away. Unfortunately this is not always reliable in practice.

TABLE 1: Constant-timeness of generated code for `ct_select_u32` with boolean condition `bool b` for different Clang versions. ✓ indicates the code generated is branchless; ✗ indicates the opposite.

		VERSION_1		VERSION_2		VERSION_3		VERSION_4	
		inlined	library	inlined	library	inlined	library	inlined	library
Clang 3.0	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✗
	-O2	✓	✓	✓	✗	✗	✓	✓	✗
	-O3	✓	✓	✓	✗	✗	✓	✓	✗
Clang 3.3	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗
Clang 3.9	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗

Attempts at constant-time coding are presented in Listing 2. The source code of `ct_select_u32()` is carefully designed to contain no branch. This sort of obfuscated code is used in smartcard implementations and in widely-used cryptographic libraries. We tested each of the four versions (annotated VERSION_1 to VERSION_4 in Listing 2) by compiling them with options `-m32 -march=i386` for `clang-3.0`, `clang-3.3`, `clang-3.9` and for different optimization levels `-O1, -O2, -O3`. We then looked at

```
int ct_isnonzero_u32(uint32_t x) {
  return (x|-x)>>31;
}
uint32_t ct_mask_u32(uint32_t bit) {
  return -(uint32_t)ct_isnonzero_u32(bit);
}
uint32_t ct_select_u32(uint32_t x, uint32_t y,
    bool bit /*={0,1}*/) {
  // VERSION_1
  uint32_t m = ct_mask_u32(bit);
  return (x&m) | (y&~m);

  // VERSION_2. Same as VERSION_1 but without
  // using multiple functions
  uint32_t m = -(uint32_t)((x|-x)>>31);
  return (x&m) | (y&~m);

  // VERSION_3
  signed b = 1-bit;
  return (x*bit) | (y*b);

  // VERSION_4
  signed b = 0 - bit;
  return (x&b) | (y&~b);

}
```

Listing 2: Constant-time selection of x or y.

TABLE 2: Constant-timeness of generated code for `ct_select_u32` with integer condition `uint32_t b` for different Clang versions. ✓ indicates the code generated is branchless; ✗ indicates the opposite.

		VERSION_1		VERSION_2		VERSION_3		VERSION_4	
		inlined	library	inlined	library	inlined	library	inlined	library
Clang 3.0	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✓
	-O2	✓	✓	✓	✓	✗	✓	✓	✓
	-O3	✓	✓	✓	✓	✗	✓	✓	✓
Clang 3.3	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✓
	-O2	✓	✓	✗	✓	✗	✓	✗	✓
	-O3	✓	✓	✗	✓	✗	✓	✗	✓
Clang 3.9	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✓
	-O2	✓	✓	✗	✓	✗	✓	✗	✓
	-O3	✓	✓	✗	✓	✗	✓	✗	✓

the code generated for each version of the function *1*) in a shared library, and *2*) when inlined in the caller (e.g. typically the case if the function is defined and called in the same module).

We present the findings in TABLE 1. First, not all optimization levels generate constant-time code: the compiler sometimes introduces a jump (refer to Listing 3 on the following page for output). We also note that as the compiler version increases (Clang 3.0, 3.3 and 3.9), more implementations become insecure (i.e. non-constant time): this is because compilers become better at spotting and optimizing new idioms. This is a real problem because code that is secure today may acquire a timing channel tomorrow. More interestingly, we observe differences if we use a single function (VERSION_2) or split its code into several smaller functions (VERSION_1). For more traditional C code, this is never a problem; but when considering side effects, even the smallest of changes can matter. There are also noticable differences in shared libraries vs. inlined code. Again, this is highly unpredictable and hard for a

developer to control.

One rule of thumb to further reduce the likelihood of compiler-introduced branches is to remove the use of conditional (i.e. bool) in order to obfuscate the condition bit. In the function declaration, this means replacing `bool bit` with an integer `uint32_t bit`. We found that although this improves constant-timeness of generated code, it is still not sufficient (TABLE 2 on the previous page) if the function is called with a `bool` as parameter (which the compiler implicitly casts to an integer).

So an extra layer of obfuscation used by cryptographers is to eradicate `bool` completely in critical code; and to have specially-crafted functions to compare integers in constant time too. OpenSSL currently declares 37 different functions to support this. Unfortunately, compilers offer no guarantees to such code; the next version of the same compiler may silently understand it and optimize the constant-timeness away. Examples of such failures include the carefully-crafted constant-time implementation of curve25519 which was broken by Microsoft's compiler in 2015 [30].

The current status quo is to be very wary of side effects of compiler output. A large body of work exists on verifying constant-timeness of code [31–34]. Although these approaches are helpful, none of them considers having the compiler as an ally rather than an enemy. We advocate doing exactly this. In Section 4.1 on page 7, we will describe constant-time selection support we added to Clang/LLVM; which offers a better guarantee of the constant-timeness of generated code. The developer can use a single function (instead of juggling between the 37 in OpenSSL); this should also improve code readability and productivity. This will benefit not just widely-used cryptographic software, but also verified cryptographic implementations that also rely on controlling side effects of generated code [35, 36].

```
1
2  ; clang-[3.3,3.9] -O[2,3] -m32 -march=i386
3  ; VERSION_2
4  ct_select_u32:
5      mov     0x4(%esp),%al
6      test    %al,%al
7      jne     L       <------- ** JUMP **
8      lea     0xc(%esp),%eax
9      mov     (%eax),%eax
10     ret
11 L:  lea     0x8(%esp),%eax
12     mov     (%eax),%eax
13     ret
14
15 ; clang-[3.3,3.9] -O[1,2,3] -m32 -march=i386
16 ; VERSION_3
17 ct_select_u32:
18     mov     0x4(%esp),%cl
19     movzbl  %cl,%eax
20     or      $0xfffffffe,%eax
21     test    %cl,%cl
22     jne     L       <------- ** JUMP **
23     xor     %ecx,%ecx
24     jmp     M
25 L:  mov     0x8(%esp),%ecx
26 M:  inc     %eax
27     and     0xc(%esp),%eax
28     or      %ecx,%eax
29     ret
30
31 ; clang-[3.0,3.3,3.9] -O[1,2,3] -m32 -march=i386
32 ; VERSION_4
33 ct_select_u32:
34     mov     0x4(%esp),%al
35     test    %al,%al
36     jne     L       <------- ** JUMP **
37     lea     0xc(%esp),%eax
38     mov     (%eax),%eax
39     ret
40 L:  lea     0x8(%esp),%eax
41     mov     (%eax),%eax
42     ret
```

Listing 3: Code generated for different compilers/versions for the constant-time `ct_select_u32()` function.

2.2.2. Secret Erasure. Another interesting example of controlling side effect is the erasure of sensitive data such as crypto keys from RAM – a problem practitioners have been fighting for years [37, 38].

Erase Function:

Consider the example in Listing 4. The programmer defines a `sensitive_buffer` variable on the stack. After using it, she wishes to erase it by filling it with 0s through a call to `memset()`. In effect, she is trying to control side effects of the stack content. However, the compiler realizes that `sensitive_buffer` goes out of scope as `sensitive_function` returns, so it removes the call to `memset()`. This "dead store elimination" happens in both gcc and Clang when optimization is enabled. The current answer to this well-known problem is to use C11's `memset_s()` function, which the compiler guarantees to never remove. Because this is not widely supported by C compilers, cryptographers, again, are left fighting the compiler: they use a specially-crafted function that obfuscates the code so as to prevent its removal during compilation. OpenSSL and mbedTLS code for scrubbing memory is presented in Listing 5 on the following page. Both approaches are unsatisfactory:

```
1 void sensitive_function(...)
2 {
3   u8 sensitive_buffer[KEY_MAX] = "\0";
4   load_data(sensitive_buffer, KEY_MAX);
5   use_data(sensitive_buffer, KEY_MAX);
6   memset(sensitive_buffer, 0, KEY_MAX);
7 }
```

Listing 4: Attempt to erase a stack buffer.

- OpenSSL's implementation is overly complicated, and the generated code is less compact, and slower, than strictly needed to erase memory.
- mbedTLS's implementation uses the `volatile` keyword, which prevents the compiler from caching values in registers, and may lead to slower code too – it writes zeros one byte at a time since it uses char pointers. The Linux kernel has an `ACCESS_ONCE` macro that casts a variable to `volatile` only for specific checks, but this relies on a particular compiler implementation to work, and led to bugs with `gcc-4.6` and `gcc-4.7` [39].

```
// code for OpenSSL's OPENSSL_cleanse
unsigned char cleanse_ctr = 0;

void OPENSSL_cleanse(void *ptr, size_t len)
{
    unsigned char *p = ptr;
    size_t loop = len, ctr = cleanse_ctr;
    while (loop--) {
        *(p++) = (unsigned char)ctr;
        ctr += (17 + ((size_t)p & 0xF));
    }
    p = memchr(ptr, (unsigned char)ctr, len);
    if (p)
        ctr += (63 + (size_t)p);
    cleanse_ctr = (unsigned char)ctr;
}

// code for mbedTLS
void mbedtls_zeroize( void *v, size_t n ) {
    volatile unsigned char *p = v;
    while( n-- ) *p++ = 0;
}
```

Listing 5: `secure_erase()` implementation for OpenSSL and mbedTLS.

- There is no strong guarantee that a compiler will not recognize these idioms and optimize them somewhat. For example, a discussion among LLVM developers concluded that it was valid to remove volatile stores if the address is a stack allocation that does not escape and the result of the store can be proven never to be read [40].

Many of these techniques depend on the compiler remaining ignorant. For example, if the scrubbing function is declared in one compilation unit and called in another, a compiler is not able to inline it and misidentify it as a `memset` that it can elide. Unfortunately, compilers increasingly make use of Link Time Optimization (LTO) (e.g. for data flow analysis). It may therefore become aware of the semantics of `secure_erase()`. LTO typically provides a 5-10% performance improvement for little cost [41] and so is increasingly the default for large software projects, but defeats many approaches that rely on hiding information about the implementation of a function from the compiler. Of course, one could separate compilation units to avoid LTO, but this is just another example of fighting the compiler rather than having it as an ally.

Erasing Stack: In addition to the problems mentioned above, the `secure_erase()` function is not sufficient for a programmer to properly scrub memory, because it does not allow clearing of state that is not part of the C abstract machine. In other words, even if `secure_erase()` is not removed and it successfully erases memory intended by the programmer, there are side effects of generated code it cannot handle. The compiler may generate code that temporarily copies parts (or all) of a buffer content to other stack areas known as "stack slots". These do not correspond to variables declared in the original source code so a developer cannot erase them programmatically. This often happens because of register pressure; and the compiler must temporarily "spill" registers to the stack. More generally, this also happens before function calls (e.g. if a register is caller-saved, it is pushed on the stack) and in function prologues (e.g. a callee-saved register is temporarily saved on the stack). Regardless of the use of `memset_s()` or a specially-crafted `secure_erase()`, a compiler offers no guarantee about "leaks" of secret data to the stack. This could be particularly problematic if the implementation uses extended registers (128 to 512 byte wide), as is often the case for AES.

To test the scope of this problem in practice, we implemented a runtime taint-tracking engine tool (available at [42]) on top of Valgrind to follow tainted (i.e. secret) data during program execution. We tested this on OpenSSL, mbedTLS and GnuPG (RSA and AES operations only). We tested programs rather than just functions. This is made relatively easy because most cryptographic libraries contain unit tests (mbedTLS) or a versatile command line interface (OpenSSL and GnuPG). We found that formatting functions (e.g. `*printf`, `*scanf`, etc.) are prone to leaving residual data on the stack. We also found that recursive functions tend to leave residual data, as they aggressively spill registers on the stack. And more generally, a lot of data was left unerased on the stack due to the Application Binary Interface (ABI), calling conventions, and register spills. Most of the time, these were short values (e.g. 64bits on x64); but as we mentioned earlier, extended registers could have more serious implications. So even these specially-crafted erasure functions do not provide high guarantees in practice.

The only chance practitioners have today to erase the stack is to try to outwit the compiler. Let us consider a candidate solution as presented in Listing 6 on the next page: the programmer declares a `currentStack` variable to obtain the current stack pointer, then it uses it to erase the stack through a call to `secure_erase()`. This assumes:

1) The compiler allocates variable `currentStack` at the bottom of the stack (on x86).
2) The compiler allocates all local variables on the stack, not in registers.
3) Local variables are contiguously allocated, with no padding between them.
4) The call to `secure_erase()` is not optimized away.

All these assumptions may be violated in practice. A compiler can allocate variables anywhere on the stack, pad them for alignment purposes, spill registers (including extended ones) on the stack to re-use them, push registers on the stack according to calling conventions, etc. As we described in Section 2.2 on page 2, the call to `secure_erase()` may also be removed.

3. Side Effects in Cryptography

Compilers optimize out a wide range of implicit code properties intended by cryptographers. In the previous section, we presented two detailed motivating examples. In this section, we provide more general use cases that would benefit from controlling side effects.

Constant Side Effects: The most common countermeasure against side-channel attacks is to make all side effects constant: constant power drain, constant electromagnetic (EM) emanations, constant-time executions, etc. We al-

```
1
2  extern int foo2(int, u8*);
3  extern int foo3(int, int, u8*);
4  extern void secure_erase(u8*, size_t);
5
6  #define FUNCTION_STACK ( sizeof(a)+sizeof(b)+
       sizeof(ret)+sizeof(buf)+sizeof(currentStack)
       )
7  #define ARG_CALLEES ( sizeof(a)+sizeof(b)+sizeof(
       u8*) )
8  #define STACK_SIZE ( FUNCTION_STACK + ARG_CALLEES
       + sizeof(currentStack) )
9
10 int foo(int c, int d)
11 {
12   int a, b, ret;
13   u8 buf[256];
14
15   a = c;
16   b = d;
17
18   ret = foo2(a, buf);
19   if (ret != 0) return ret;
20
21   ret = foo3(a,b,buf);
22   if (ret != 0) return ret;
23
24   [...]
25
26   // try to erase the stack
27   int currentStack;
28   secure_erase(((u8*)&currentStack), STACK_SIZE);
29
30   return ret;
31
32 }
```

Listing 6: Erasing stack by outwitting a compiler, x86.

ready elaborated on the need for constant-time side effects; the same basic ideas apply to other side effects too.

Other compiler threats include code motion, common subexpression elimination (CSE), and peephole optimizations [43]. Constant-time arithmetic needs not just constant-time comparison [44] and conditional swap [45] but attention to whether operation timings depend on operand content [46]. Changes to instruction ordering might leave timings unchanged but have an effect on power consumption, making differential power attacks easier. So the desired security properties are not just about stopping dead code elimination in particular functions; they may require support from many layers of the stack.

Secret-Independent Control Flow: Microsoft's compiler introduced a secret-dependent jump in the constant-time implementation of curve25519, leading to the recovery of key material remotely [30]. A toolchain with explicit controls of side effects might not only guarantee a programmer's invariant, but also warn the programmer if a sensitive variable affects the control flow. This would often be understood quickly to be a mistake.

Secret-Independent Memory Access: The same holds for memory access. For example, many AES implementations have table lookups indexed by secret data, and may use random values (blinding) to prevent an attacker exploiting cache-based side channels by polluting the cache [3, 47, 48].

Secret-Independent Loop Bounds: Secret-dependent loop bounds may also leak sensitive information during operations such as padding validation, group addition, or simply during memory comparison.

Noise Addition: Programmers often want to add noise to a program to mitigate information leakage via side channels (e.g. acoustic, power, EM, time). They may add random delays and arbitrary code [49, 50]; run different codes of the same algorithm depending on when it is executed; or shuffle instructions randomly [51], etc. But as the side channels are side effects, there is a risk that a compiler will optimize away the noise too.

Bit Splitting scatters sensitive data across memory to make it more difficult for an attacker to locate and reconstruct sensitive data from a RAM dump or chip scan. Coded by hand, this might amount to using eight 1-bit variables instead of a single 1-byte variable. But this is tedious to do by hand, and may be optimized away by the compiler.

Bit Splicing also splits variables into single-bit variables and then uses only single-bit logical operations (e.g. `AND`, `OR`, `XOR`, `NOT`, etc.). This helps fight timing side channels by removing conditions. Again, this may be optimized away by the compiler.

Fault Injection Countermeasure: An attacker may use a laser or a rowhammer technique to inject memory faults. Cryptographers sometimes read variables multiple times from various locations to spot inconsistent changes. But this also gets optimized away by the compiler. One possible fix is to declare those variables as `volatile`; but this slows down the program, especially if the value is a global. Compiler support for countermeasures against changes in critical code and data would be welcome.

Secret Erasure: Scrubbing sensitive data from RAM is a problem with which cryptographers have been struggling for years [37, 38, 52, 53]. We illustrated this problem in Section 2.2 on page 2.

4. Implementations of Explicit Side Effects

Previous sections demonstrated the need for compiler support to control side effects of security-sensitive code. Modern compiler IRs differ, but the core requirements for implementing such features are present in all of them. In this section, we present the support we added in Clang/LLVM for both constant-time selection and secure stack erasure from Section 2 on page 2. We argue that this should be only the first step in a sustained engineering effort to provide explicit compiler support for implicit properties.

Clang/LLVM is a compiler framework divided in three core parts: *1*) the frontend which reads the source code (for C, this is Clang); *2*) LLVM where all common optimizations take place (e.g. dead-code elimination, common subexpression elimination, etc.); and *3*) the backend, where target-specific optimizations take place (e.g. peephole optimization) and machine instructions are emitted. Well-known optimizations responsible for side effects are caused by LLVM (e.g. dead code elimination may remove the call to zero a buffer). But backend implementations may also get in the way: for example, if we compile the naive select of Listing 1 on page 3 with `clang-3.9`

`-m32 -march=i386`, then the LLVM representation is branchless; but the generated binary no longer is. This remains true even if we replace the condition `bool b` by an integer `uint32_t b`, and for any optimization level `-O0`, `-O1`, `-O2`, `-O3`. Thus, to have high guarantees about side effects, we must avoid both LLVM and the backend optimizations.

For all evaluations reported in the rest of this paper, we run programs on a server installation of Debian Jessie. To minimize the impact of other programs running on the machine, we boot it with kernel option `init=/bin/bash`. This makes the kernel start `/bin/bash` instead of the usual init daemon. This, among other things, ensures no additional services are started, including networking.

4.1. Constant-Time Selection

4.1.1. Implementation. As we explained in Section 2.2.1 on page 3, OpenSSL currently defines 37 functions to implement constant-time selection. Even though this seems to work on today's compilers, there is no garanatee that the next generation will not break the constant-timeness. As compilers become smarter, they become better at spotting idioms and at optimizing them (Section 2.2.1 on page 3).

So we implemented a unique builtin function `__builtin_ct_choose(bool cond, x, y)` in the Clang/LLVM framework (a reference implementation is available at [8]). In the frontend, we automatically detect the integer type of `x` and `y` so we do not need a different function for every different type. If their types do not match, the compiler outputs an error and refuses to compile. We purposely abstained from supporting floating point integers as these are known to lead to non-constant execution times [54].

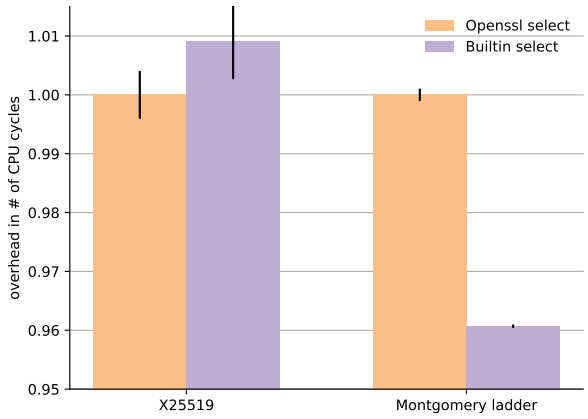
The `__builtin_ct_choose` function we implemented can be called with the condition `bool b` expressed with the usual comparison operators (e.g. >, ==, !=, etc.). It does not require the specially-crafted comparison functions that modern cryptographic libraries use. This improves code readability, API usability and developer productivity. To ensure the compiler does not create a branch depending on the condition, we mark the function as "not duplicable" throughout its life cycle (*IntrNoDuplicate* in LLVM, *isNotDuplicable* in the backend).

We added custom support to the x64 backend by ultimately compiling the function into a `CMOV` after other optimizations have been performed. This way we avoid backend optimizations too. `CMOV` was empirically shown to be constant time by previous work [55, 56]. We did not aim at challenging this assumption in this work. A compiler may replace this with a `XOR` or other constructs, without source code modification if it is found to be invalid. For other backends that do not provide custom support for constant-time selection yet, we implemented a default one that compiles `__builtin_ct_choose` to a `XOR` instruction (VERSION_1/2/4 in Listing 2 on page 3) – the `XOR` instruction is more likely than multiplication (VERSION_3) to be constant-time for different operands. The default implementation happens at the start of the backend so it avoids optimizations of LLVM but not the backend ones. This is still better than any hand-crafted version that goes through the LLVM optimizations, as we show next.

4.1.2. Evaluation. For x64, we verified that generated code did indeed contain the expected `CMOV` instructions for various programs including OpenSSL and mbedTLS. For other backends, we focused on the code we presented in Listing 2 on page 3; and that we showed was often compiled into non-constant-time code by Clang. We found that with our compiler solution, and even though the default-backend implementation does not avoid backend optimization, the generated binary contained no branches. For x64 and x86, we further tested the constant-time implementations with two crypto algorithms that make use of multi-variable selection: OpenSSL's X25519 (the Diffie-Hellman exchange that uses curve25519) and a self-written constant-time RSA exponentiation using the Montgomery ladder [57]). We used a tool called Dudect [33]: this runs the program with millions of different inputs to detect significant time differences indicative of non-constant time code. Constant-time code is hard to come by, so testing a large corpus of constant-time code was deemed out of scope of this work.

Wide adoption of security primitives is hampered by the non-negligible overhead they usually incur. So we benchmarked the two crypto implementations too. For each, we compiled with the builtin `__builtin_ct_choose` and with OpenSSL's hand-crafted version, then ran them 100 times. The results are presented in Figure 1. For X25519, our builtin solution incurs less than 1% running time overhead; for the Montgomery ladder our builtin solution is about 4% faster.

Figure 1: Runtime overhead for X25519 (OpenSSL) and RSA Montgomery ladder.



4.2. Stack and Register Erasure

We saw in Section 2.2.2 on page 4 that it is impractical for a developer to programmatically erase sensitive data spilled on the stack. So our goal in this section is to provide explicit support for this in a mainstream C compiler, through function annotations. For example, to

enable instrumentation for a function `foo()`, a programmer simply precedes its declaration and definition with the `__zero_on_return` keyword.

Right before the function returns, our instrumentation zeros the stack and registers used. It is important to zero the registers too as they may be spilled to the stack later. We opted for Clang/LLVM as the compiler framework to implement our solution (a reference implementation is available at [9]); x64 Linux as the target platform; musl-libc [58] as the standard C library and runtime linker; and the gold linker as static linker. As a proof-of-concept, the musl-libc run-time linker was easier to modify. Our approaches should work with any C implementation, with varying amounts of engineering effort. Unlike current approaches discussed in Section 2.2.1 on page 3, our solution uses no source-code tricks: we implemented our instrumentation in the compiler backend after other backend optimizations have taken place. This gives strong guarantees that the code is not altered.

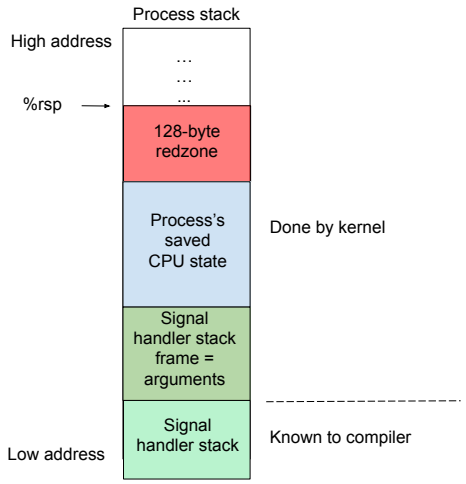
In the rest of the paper, we assume a single-threaded userspace program P that uses secret data at time T_{use}. Once P is done using the secret, it tries to erase it. The erasure is secure, or successful, if an attacker with access to the virtual address space of P at a time $T_{attacker} > T_{use}$ gains no information that helps her recover the secret (key or plaintext). We do not consider adversaries with physical access to the memory. We assume the kernel does not leak secrets.

Given the large amount of code involved in modern toolchains (the OS kernel, libraries, loader, linker, the ABI, the compiler and the programmer), we abstain from trying to formally verify our implementation. We see value in a working implementation that we can open source today and we leave formal guarantees for future work. Next, we detail what is really involved to implement proper erasure in a mainstream C compiler today, and the assumptions we make throughout.

4.2.1. CPU, OS and ABI. A userspace program does not run in isolation, but on a machine providing its own set of libraries. There is a lot of platform code which we cannot instrument when we compile a userspace application alone. As well as libc and the runtime loader/linker, we have to consider the Virtual Dynamic Shared Object (VDSO), a piece of executable code provided by the OS kernel that is mapped into userspace as a program starts. If we want to ensure erasure of the stack, we must recompile all of these.

Even a single-threaded application can receive asynchronous signals in Linux. When this happens, the kernel stops one of the application's threads of execution and executes the signal handler. Prior to this, though, the kernel saves the current program's CPU state (i.e. registers) on the userspace stack. If a program is in the middle of a sensitive function, these registers are likely to contain sensitive data. So we must erase the part of the stack where the kernel saves them – on x86, where the application's current `%esp` points and on x64, 128 bytes below it (Figure 2). This 128-byte memory region is referred to as the "red zone" and is guaranteed by the x86-64 System V ABI to never be clobbered by the kernel during signal handling, allowing leaf functions to spill values to the stack without having to move the stack pointer. So we must account for this extra 128-byte offset when calculating the amount of stack to erase.

Figure 2: Stack layout during signal handling on x64.



4.2.2. The Compiler and the Linker. Like any modern software, Clang/LLVM is made up of several modules. We must be aware of how these fit together to implement our solution.

Compilers use a runtime library of architecture-dependent optimized functions to speed up certain operations, e.g. `__udiv*` for division. For LLVM, this is called *compiler-rt*; while for gcc it is *libgcc*. We must recompile these too.

Besides the compiler runtime, a compiler may also substitute calls to known functions with its own optimized version, inlined for performance. This is typically the case for calls to libc's memory-intensive functions such as `memset()`, `memcpy()`, etc. in LLVM.

Certain compiler optimizations may change the usage of registers. For example, `%ebp` is typically used as frame pointer, so it stores a (non-sensitive) address. If the compiler realizes a function does not need a frame pointer (e.g. a leaf function), it may use `%ebp` to store sensitive data instead. We must erase the resgiter in the latter.

A number of compiler optimizations also affect the stack. Tail-call optimization reduces stack accesses during function calls. Let us consider the pseudocode of Listing 7 on the following page, which defines three functions: `function_A` that calls `function_B`, `function_B` that calls `function_C`, and a leaf function `function_C`. Before each function returns, there is an `__erase_stack` which represents a piece of code that erases the stack (described in next section). Now consider Listing 8 on the next page representing the same code with tail-call optimization: `function_B`'s `call` to `function_C` is now replaced with a `jmp` (Line 9), so no return address is pushed to the stack. This means that when `function_C` returns (Line 18), the only return address on the stack is the one from `function_A`, and so

`function_C` returns directly to `function_A`, thereby skipping the `__erase_stack` present in `function_B` (Line 11). So if we are not careful, the code intended to erase the stack may be skipped.

Defer-pop optimization is a gcc optimization [59] that "lets arguments accumulate on the stack for several function calls and pops them all at once." This also changes the stack layout, and may have implications for stack usage. It is not supported in LLVM at the moment.

Shrink-wrapping optimization also tries to reduce stack usage (Listing 9). Without optimization, all callee-saved registers that may be used in a function are pushed to the stack (Lines 3-7). With shrink-wrapping optimization (Listing 10), registers are pushed only if they are certain to be used (e.g. within an `if-statement`, Lines 4-6 and Lines 19-20). This affects the stack layout.

Function Multiversioning [60] is a gcc feature that selects the best function version at runtime, depending on the target. This is not supported by LLVM. The dispatching mechanism will have to be instrumented as well in future LLVM versions.

The last step to generating a binary is for the static linker to link all objects together. To link against shared libraries (e.g. libc), the static linker adds stubs to redirect function calls to the runtime linker at runtime. For additional guarantees, these stubs should also be instrumented.

```
1  function_A:
2    ...
3    call function_B
4    __erase_stack
5    ret
6
7  function_B:
8    ...
9    call function_C
10   __erase_stack
11   ;; this returns to
       function_A
12   ret
13
14 function_C:
15   ...
16   __erase_stack
17   ;; this returns to
       function_B
18   ret
19
20
```

Listing 7: Non-optimized code.

```
1  function_A:
2    ...
3    call function_B
4    __erase_stack
5    ret
6
7  function_B:
8    ...
9    jmp function_C
10   ;; this is never
       called
11   __erase_stack
12   ret
13
14 function_C:
15   ...
16   __erase_stack
17   ;; this returns to
       function_A
18   ret
19
20
```

Listing 8: Tail-call optimized code.

```
1  foo:
2    ; all registers are
       pushed
3    push r1
4    push r2
5    push r3
6    push r4
7    push r5
8
9    if r0:
10     r1 = ...
11     r2 = ...
12     r3 = ...
13   else:
14     r4 = ...
15     r5 = ...
16
17   ; all registers are
       popped
18   pop r5
19   pop r4
20   pop r3
21   pop r2
22   pop r1
23
24
```

Listing 9: Non optimized code.

```
1  optimized_foo:
2    if r0:
3      ; only used
         registers are
         pushed
4      push r1
5      push r2
6      push r3
7
8      r1 = ...
9      r2 = ...
10     r3 = ...
11
12     ; only used
         registers are
         popped
13     pop r3
14     pop r2
15     pop r1
16
17   else:
18
19     push r4
20     push r5
21
22     r4 = ...
23     r5 = ...
24
25     pop r5
26     pop r4
27
28
```

Listing 10: Shrink-wrapping optimized code.

4.2.3. The Programmer. As always, we also need cooperation from the developer to achieve proper erasure.

For examples, certain functions, such as `exit()`, `assert()`, `abort()`, `execve()`, etc. never return. As we erase stack and registers before a function returns, we cannot erase them for non-returning functions. In theory, we could support non-returning functions by erasing all stack and registers before the call. We would have to be careful not to erase stack area and registers passed as arguments to such functions. For `exit(int)` and `abort(void)`, this should be fairly simple. But in general for other functions, one would have to be extremely careful to get this right. So for our reference implementation, we decided against adding complexity, and we implicitly rely on the kernel to zero those pages when the process terminates. This gives different guarantees whether the kernel erases these pages straightaway or lazily at the time of re-use. A developer should avoid calling such functions in sensitive code.

The use of variable-sized stack objects does not lend itself to low-overhead compile-time stack erasure approaches, as the compiler cannot determine stack usage at compile time. These are used rarely (we found only two occurrences in our tests) and can simply be replaced with a call to `malloc()`. In a security context, using variable-sized stack objects should be avoided anyway, as it may allow an attacker to cause stack overflows [61].

Certain libc functions affect the stack location, e.g. `sigaltstack()` allows a programmer to change the location of the stack used by signal handlers. We decided not

to add unnecessary complexity to support such functions since these are not used in crypto code.

To ensure a developer does not inadvertently break the above assumptions, our compiler solution will refuse to compile if any of these problematic corner-cases are present in the source code.

4.2.4. Implementation and Evaluation. We implemented three different solutions. The first, naïve one is function-based (**FB**): it performs the erasure for every sensitive function and its callees. This is not optimal: callees will erase the same stack area repetitively. So we introduce a stack-based solution (**SB**) where callees only keep track of stack usage. Then, only the sensitive function does the erasure, and only once. Keeping track of stack usage requires only register operations, which is faster than performing erasure. The **SB** solution shows just 1% runtime overhead in our tests. However the size of the executable increases as a result of the instrumentation (by less than 10% in our tests). So for systems that require more compact code, we explore a third solution based on the call graph (**CGB**): the call graph is used to determine the maximum stack usage of a sensitive function at compilation time, so it eliminates callee instrumentation. We describe each solution in more details, next.

Function-Based Solution (FB):

Before an annotated function returns, this solution erases the registers and stack it has used. This is identical to what we unreliably tried to express in Listing 6 on page 6. Special care must be taken for returned registers because the upper bits of their "parent register" may contain sensitive data: for example if register `%eax` is returned, we erase the 32 upper bits of `%rax`. We do not currently handle non-returning functions such as `exit()`. We implicitly rely on the kernel to do this. Tail call optimizations must be disabled for annotated functions, as they turn returning functions into non-returning ones – in our tests we simply disabled this globally with compiler flag `-fno-optimize-sibling-calls`. This would be done per function for a production build.

We did not instrument the PLT stubs introduced by the static linker because musl-libc does not support lazy binding so all symbols are resolved at load time. Therefore the stubs simply jump to the resolved functions at runtime and no spill to the stack happens.

We implemented two sub-solutions for the **FB** solution: one with support for signal handlers (**FB with SH**), and one without (**FB no SH**). There are at least two reasons why a programmer would not want signal support: *1*) if the program does not catch signals, or *2*) if the kernel is patched to zero the position on the stack where it saves the program's state.

We verified that functions in compiler-rt, OpenSSL and mbedTLS contained our instrumentation after compiled with our pass. The adoption of security features is affected by their overhead, so we also used the MiBench benchmark [62] to see the overhead incurred[1]. For this, we instrumented all the functions in musl-libc, compiler-rt and the MiBench programs. This required a mere re-compilation, and no code changes were needed. We ran each program 30 times to obtain the results of Figure 3 on the next page.

Programs instrumented with signal handler support (**FB with SH**) run 3.39 times slower, whereas those without (**FB no SH**) run 1.86 slower. The reason solution **FB with SH** is much worse is because we must assume a signal is received in every function, so there is an additional per-function cost. We study a faster solution next.

Stack-Based Solution (SB):

In this solution, we instrument all functions to keep track of runtime stack usage through a global variable `__GlobalStackValue` we added in musl-libc. When an annotated function returns, we erase the stack used by itself and its callees, as reflected by the value of `__GlobalStackValue`. In our current implementation, we made the choice to instrument functions after the epilogue (before the `ret` instruction). It is possible to move this code to the prologue; but we chose the former solution because at the end of a function, most registers are no longer live so we can re-use them without spilling them temporarily to the stack. As a result of this choice, we require functions to have an epilogue and return, hence we disable tail-call optimization. Ideally, we would only disable it in places where it would affect our security guarantees, rather than globally. But this has little impact on speed anyway as we show next.

We implemented two sub-solutions: *1*) one with "bulk register zeroing" (we zero all registers at once in annotated functions only – **SB with BRZ**), and *2*) one where we erase registers in every function individually (**SB no BRZ**). Note that for **SB with BRZ**, we do not keep track of registers, so we assume they are all clobbered and erase them all – except of course for those explicitly returned by the annotated function.

We verified that functions in compiler-rt, OpenSSL and mbedTLS contained our instrumentation after compiled with our pass. As we want this work to be deployed on real systems, we care about the performance overhead incurred. We found these implementations improved performance significantly (Figure 3 on the following page) compared to the **FB** solutions, with just 2% overhead for **SB no BRZ**, and 1% for **SB with BRZ**. The stack-based erasure solution (**SB**) is faster than the function-based one (**FB**) because in the former, we only perform operations on registers (which is fast) to keep track of stack size; in the latter we erase memory in every function which is slower (access to RAM or the CPU caches). Two outliers (`CRC32` and `bitcount` benchmarks) run slower: those contain tight loops calling functions where our instrumentation dominates. For example, `CRC32` calls musl-libc's `puts()` which is rather short, and the call is not inlined: so on every `puts()`'s return our instrumentation incurs some overhead. The size of the musl-libc's shared library increased by 6.6% and 4.9% for **SB no BRZ** and **SB with BRZ** respectively, which we think is acceptable for most platforms.

Next, we present a more compact implementation suitable for embedded devices where code size matters more.

[1]We selected MiBench over SPEC CPU2006 because the former is free

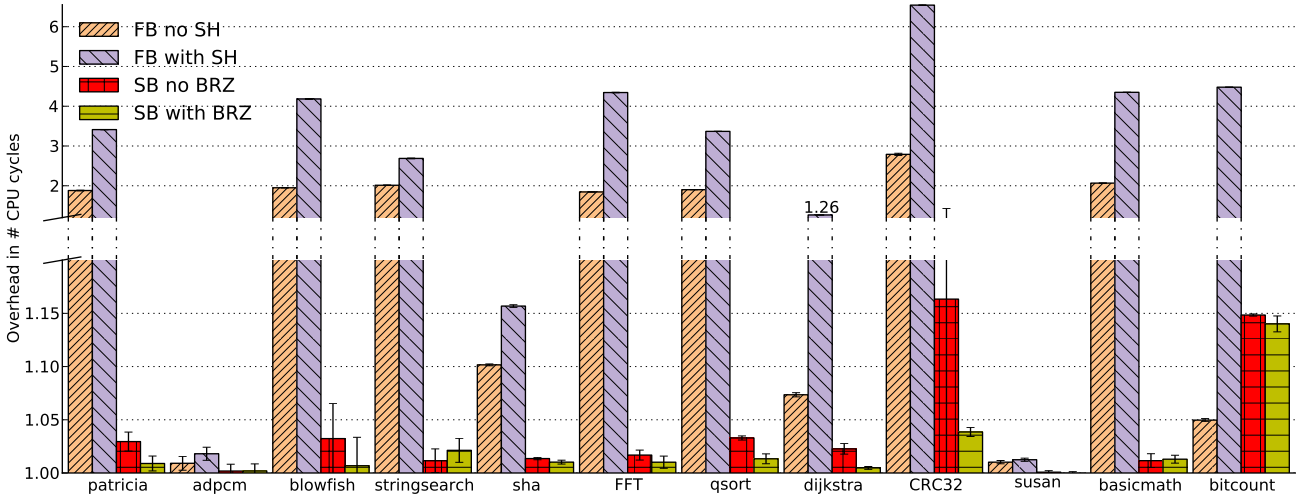


Figure 3: Runtime overhead for MiBench programs. The call graph solution (**CGB**) is omitted because there is no instrumentation (i.e. no additional overhead) besides the actual zeroing of the stack and registers.

Call-Graph Based Solution (CGB):

In this solution, we leverage the call graph known at compilation time. The call graph allows us to determine the maximum stack usage any path of an annotated function will ever use: this removes the need to instrument every function, which is faster. We also use the call graph to obtain the list of registers that may be written to. Before an annotated function returns, we erase the stack and the registers. This is the fastest and most compact solution as we only instrument annotated functions, and not its callees. We do the minimum amount of work and the only performance overhead is down to the erasure itself, which is inevitable – this is why we omit it in Figure 3.

Tail call optimization is compatible with this solution, except for (the few) annotated functions. To support function pointers, we ask programmers to tag them as well as the functions they may point to (an example is presented in Listing 11 on the following page). This gives us a set of candidate functions we can use at compilation time to compute the call graph. To reduce the number of annotations a developer has to write, our solution allows annotating `typedef`'ed function pointers: this way any declaration of such pointer automatically inherits the annotation. At compilation time, we also validate that pointer usage respects annotations: compilation will fail if the developer uses a non-annotated pointer where an annotated one is expected, e.g. in an assignment.

During the computation of the call graph, a general implementation would normally account for lazy binding, which redirects execution to the runtime linker the first time a function is called. However, because musl-libc does not support lazy binding and always resolves symbols at program startup, this was not necessary in our case.

The **CGB** approach does not lend itself to cycles in the call graph, as these create infinite loops. So recursive functions should be avoided within sensitive code. In musl-libc, we found certain functions to be recursive but with a maximum recursion depth: these can be handled simply by multiplying their stack usage by this maximum to obtain a conservative bound on their runtime stack.

Code using asynchronous events (e.g. POSIX asynchronous I/O (AIO) [63]) should avoid accessing sensitive data. This is because these APIs lead to a non-deterministic call graph that cannot be reliably determined at compilation time. This holds true for signal handlers too.

We summarize each of our three solutions' pros and cons in TABLE 3.

TABLE 3: Comparisons of Implementations. The first symbol indicates if a solution theoretically supports a feature; the second symbol indicates if our current implementation implements it. ✓indicates support; ✗no support.

	lazy binding	signal handling	call tail optimization	call graph cycles	Asynchronous APIs	variable-size stack objects	function pointers	PLT stubs
FB	✓– ✓	✓– ✓	✓– ✗	✓– ✓	✓– ✓	✗– ✗	✓– ✓	✓– ✗
SB	✓– ✓	✓– ✓	✓– ✗	✓– ✓	✓– ✓	✗– ✗	✓– ✓	✓– ✗
CGB	✓– ✗	✓– ✓*	✓– ✓**	✗– ✗+	✗– ✗	✗– ✗	✓– ✓++	✓– ✗

*Not the stack of signal handler itself.
**Except for annotated functions.
+Support for bounded recursion is possible.
++Only for annotated functions.

5. Related Work

Previous work on controlling side effects of code considers the compiler an enemy the programmer has to fight, so most research has looked at verifying implicit code properties after compilation. There has been work to detect cache side channels through static code analysis [31, 64], dynamic analysis [32, 33], and via transactional memory at runtime [65]. WYSINWYX [66] highlights the dangers in assuming that code in a compiled language has a well understood mapping to generated code.

The Software Analysis Workbench (SAW) [34] provides tools to formally verify certain properties of cryptographic C and Java code. Ct-verif [67] formally verifies

```
1
2  // definitions of tags
3  #define __TAG_HASH_INIT __attribute__((
       type_annotate("tag_hash_init")))
4  #define __TAG_HASH_UPDATE __attribute__((
       type_annotate("tag_hash_update")))
5  #define __TAG_HASH_FINISH __attribute__((
       type_annotate("tag_hash_finish")))
6
7  // tag each function pointer with the
       corresponding tag
8  struct hash_t {
9    __TAG_HASH_INIT void (*init)(void);
10   __TAG_HASH_UPDATE void (*update)(const u8*,
       size_t);
11   __TAG_HASH_FINISH void (*final)(u8*);
12 }
13
14 // define function with corresponding tags
15 // for sha3 hash functions
16 __TAG_HASH_INIT void sha3_init(void) { ... }
17 __TAG_HASH_UPDATE void sha3_update(const u8* in,
       size_t len) { ... }
18 __TAG_HASH_FINISH void sha3_final(u8 * out) { ...
       }
19
20 // define function with corresponding tags
21 // for sha256 hash functions
22 __TAG_HASH_INIT void sha256_init(void) { ... }
23 __TAG_HASH_UPDATE void sha256_update(const u8* in
       , size_t len) { ... }
24 __TAG_HASH_FINISH void sha256_final(u8 * out) {
       ... }
25
26 void heneric_hash(u8 * in, size_t len, u8 * out,
       struct hash_t * HT) {
27
28   if ( HT && in && len ) {
29
30     *(HT->init);
31     *(HT->update)(in, len);
32     *(HT->final)(out);
33   }
34 }
```

Listing 11: Annotation for function pointers.

the constant-timeness of certain components of widely-used cryptographic libraries. A verified TLS implementation was showcased by miTLS [35]. Its successor, project Everest [36], is an ongoing project to build a verified https stack. MiTLS and Everest are written in functional languages. The FaCT language [68] provides a better substrate for code with strong constant-time requirements. We would be happy if the world adopted languages with such guarantees but unfortunately software, particularly C software, has a tendency to become security critical long after it was initially written. There are also verified implementations in assembly, such as for curve25519 [69].

Copens *et al.* [55] and Molnar *et al.* [70] explored possible code transformations to combat side channels. Chapman [71] studied coding styles to reduce programmers' mistakes when sanitizing memory in Ada and SPARK.

There is notable work on making security guarantees explicit in programming languages, such as proof-carrying-code [72] and the functional language F* [73]; but not in any mainstream language like C. There has also been work embedding low-level code in F* [74], writing a verified program that generates C. This provides an attractive model for writing the code, but then relies on the C compiler not violating any of the guarantees that the programmer intended and would complement our work.

Closer to our work is a recent paper by D'Silva *et al.* [75] who present compiler optimizations violating security guarantees of source code. The CompCert compiler has been used to verify the absence of side channels from cryptographic code, though this approach relies on a formal proof of a side-channel free algorithm as input.

Other recent work on securing aspects of compiler transforms [76–78] is complementary to ours. We encourage programmers to express their security intentions in a way that allows the compiler to ensure that the required invariants are met, whereas they focus on guaranteeing that the compiler does not violate these guarantees. There are also tools, such as Vale [79] that aim to verify assembly implementations. Such tools could potentially be used for black-box verification of a compiler that supports annotations of the form that we propose.

6. Conclusion and Future Work

Many code properties that are critical to security and safety, such as the time code takes to execute, are hard for a programmer to control because of the side-effects of compiler optimizations. Previous work has tried to deal with side-effects by distrusting the compiler and verifying the generated code. However this leads to secure code being broken by compiler updates. The arms race between attacker and defender may be inevitable, but the defender should not have to fight her toolsmith as well.

We propose that compilers should explicitly support controls for implicit properties such as execution time and the zeroing of sensitive data after use. By making intentions explicit, the compiler writer can become the security engineer's ally, not her enemy. As case studies, we implemented constant-time selection and secure register/stack erasure in LLVM and showed that it can be done with minimal overhead. Our work highlights the complexity of doing this task properly, as it requires the kernel, platform libraries and programmers to synchronize their efforts. It also shows how unsatisfactory are the ad-hoc methods used up till now; only with a thorough understanding of the compiler internals can the job be done properly.

The obvious next steps are support for controlling the side-effects of sensitive data on computation, memory access and control flow; then we need to consider support for bit splitting, and for countermeasures to fault induction. Hardware-software co-design issues are next; the latest ARM designs have support for cache-line zeroing and control-flow integrity. As security and safety become one in the "Internet of Things", and the cost of code maintenance comes to dominate the total lifecycle cost of more and more things, compiler support for secure code will be essential. Defenders should be able to concentrate their efforts on stopping attackers, and the compiler writer should be an ally rather than a subversive fifth column in the rear.

7. Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and comments. This work was supported by Thales e-Security. This work is part of the CTSRD project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], and Google Inc.

References

[1] ISO, *Programming languages – C – ISO/IEC 9899:1999*. 1999.

[2] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pp. 719–732, 2014.

[3] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A timing attack on OpenSSL constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pp. 346–367, 2016.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pp. 605–622, 2015.

[5] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, "Ooh aah... just a little bit : A small amount of side channel can go a long way," in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pp. 75–92, 2014.

[6] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

[7] Y. Yarom and K. E. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," *IACR Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

[8] "Constaint-time choose for Clang/LLVM." https://github.com/lmrs2/ct_choose.

[9] "Register/Stack-zeroing for CLang/LLVM." https://github.com/lmrs2/zerostack.

[10] "C Implementation-Defined Behavior." https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/C-Implementation.html#C-Implementation. [Online; accessed 10-February-2017].

[11] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: Elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pp. 1–15, 2016.

[12] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: What happened to my code?," in *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pp. 9:1–9:7, 2012.

[13] "What Every C Programmer Should Know About Undefined Behavior." http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html. [Online; accessed 10-February-2017].

[14] "Finding Undefined Behavior Bugs by Finding Dead Code." http://blog.regehr.org/archives/970. [Online; accessed 10-February-2017].

[15] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pp. 336–345, 2015.

[16] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "A differential approach to undefined behavior detection," *Commun. ACM*, vol. 59, no. 3, pp. 99–106, 2016.

[17] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 260–275, 2013.

[18] "Signed overflow optimization hazards in the kernel." https://lwn.net/Articles/511259/. [Online; accessed 10-February-2017].

[19] "Fun with NULL pointers, part 1." https://lwn.net/Articles/342330/. [Online; accessed 10-February-2017].

[20] "Random Number Bug in Debian Linux." https://www.schneier.com/blog/archives/2008/05/random_number_b.html. [Online; accessed 10-February-2017].

[21] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 549–564, USENIX Association, 2016.

[22] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain cross-VM attacks on Xen and VMware are possible!," *IACR Cryptology ePrint Archive*, vol. 2014, p. 248, 2014.

[23] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pp. 299–319, 2014.

[24] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1155, 2015.

[25] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud," *IACR Cryptology ePrint Archive*, vol. 2015, p. 898, 2015.

[26] G. I. Apecechea, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pp. 591–604, 2015.

[27] S. Vaudenay, "Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ...," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT '02, pp. 534–546, 2002.

[28] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pp. 1–12, 1998.

[29] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pp. 526–540, 2013.

[30] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015," in *International Conference on Cryptology and Network Security*, pp. 573–582, Springer, 2016.

[31] "Comparing algorithms to track implicit flows." http://cuda.dcc.ufmg.br/flowtrackervsferrante. [Online; accessed 10-February-2017].

[32] "Checking that functions are constant time with Valgrind." https://github.com/agl/ctgrind. [Online; accessed 10-February-2017].

[33] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1123, 2016.

[34] "SAW (Software Analysis Workbench)." https://galois.com/project/software-analysis-workbench/. [Online; accessed 27-July-2017].

[35] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 445–459, IEEE, 2013.

[36] "Project Everest aims to build and deploy a verified HTTPS stack." https://project-everest.github.io/. [Online; accessed 10-February-2017].

[37] "How to zero a buffer." http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html. [Online; accessed 10-February-2017].

[38] "Secure erasure in C." https://marc.ttias.be/cryptography-general/2016-09/msg00040.php. [Online; accessed 10-February-2017].

[39] "ACCESS_ONCE() and compiler bugs." https://lwn.net/Articles/624126/. [Online; accessed 10-February-2017].

[40] "[LLVMdev] Eliminate Store-Load pair even the LoadInst is volatile." https://marc.info/?l=llvm-dev&m=121151001122373. [Online; accessed 27-July-2017].

[41] T. Johnson and X. D. Li, "ThinLTO: A fine-grained demand-driven infrastructure." EuroLLVM, 2015.

[42] "Secretgrind." https://github.com/lmrs2/secretgrind.

[43] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pp. 73–87, 2015.

[44] "Coding rules - Cryptography Coding Standard." https://cryptocoding.net/index.php/Coding_rules. [Online; accessed 10-February-2017].

[45] E. Nascimento, L. Chmielewski, D. Oswald, and P. Schwabe, "Attacking embedded ECC implementations through CMOV side channels," *IACR Cryptology ePrint Archive*, vol. 2016, p. 923, 2016.

[46] "BearSSL - Constant-Time Mul." https://bearssl.org/ctmul.html. [Online; accessed 10-February-2017].

[47] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[48] A. Canteaut, C. Lauradoux, and A. Seznec, *Understanding cache attacks*. PhD thesis, INRIA, 2006.

[49] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "A smart random code injection to mask power analysis based side channel attacks," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pp. 51–56, 2007.

[50] J. A. Ambrose, R. G. Ragel, *et al.*, "RIJID: random code injection to mask power analysis based side channel attacks," in *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pp. 489–492, IEEE, 2007.

[51] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 740–757, Springer, 2012.

[52] "MSC06-C. Beware of compiler optimizations." https://www.securecoding.cert.org/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations. [Online; accessed 10-February-2017].

[53] "Optimizer Removes Code Necessary for Security." https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537. [Online; accessed 10-February-2017].

[54] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE Symposium on Security and Privacy*, pp. 623–639, 2015.

[55] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pp. 45–60, 2009.

[56] "Compilers and constant-time code." http://www.reparaz.net/oscar/misc/cmov.html. [Online; accessed 27-July-2017].

[57] M. Joye and S.-M. Yen, *The Montgomery Powering Ladder*, pp. 291–302. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[58] "musl libc." https://www.musl-libc.org. [Online; accessed 10-February-2017].

[59] "Options That Control Optimization." https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html. [Online; accessed 10-February-2017].

[60] "Function Multiversioning." https://gcc.gnu.org/wiki/FunctionMultiVersioning. [Online; accessed 10-February-2017].

[61] "The Stack Clash." https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt. [Online; accessed 27-July-2017].

[62] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pp. 3–14, 2001.

[63] "aio - POSIX asynchronous I/O overview." http://man7.org/linux/man-pages/man7/aio.7.html. [Online; accessed 10-February-2017].

[64] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[65] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), USENIX Association, 2017.

[66] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 23:1–23:84, Aug. 2010.

[67] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 53–70, USENIX Association, 2016.

[68] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A flexible, constant-time programming language," in *Secure Development Conference (SecDev)*, IEEE, September 2017.

[69] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang, "Verifying curve25519 software," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pp. 299–309, ACM, 2014.

[70] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, (Berlin, Heidelberg), pp. 156–168, Springer-Verlag, 2006.

[71] R. Chapman, "Sanitizing sensitive data: How to get it right (or at least less wrong)," in *Ada-Europe International Conference on Reliable Software Technologies*, pp. 37–52, Springer, 2017.

[72] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 106–119, ACM, 1997.

[73] "F*: A Higher-Order Effectful Language Designed for Program Verification." https://fstar-lang.org. [Online; accessed 10-February-2017].

[74] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *PACMPL*, vol. 1, pp. 17:1–17:29, Sept. 2017.

[75] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *Security and Privacy Workshops (SPW), 2015 IEEE*, pp. 73–87, IEEE, 2015.

[76] C. Deng and K. S. Namjoshi, "Securing the ssa transform," in *Static Analysis* (F. Ranzato, ed.), (Cham), pp. 88–105, Springer International Publishing, 2017.

[77] C. Deng and K. S. Namjoshi, "Securing a compiler transformation," in *Static Analysis* (X. Rival, ed.), (Berlin, Heidelberg), pp. 170–188, Springer Berlin Heidelberg, 2016.

[78] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C.-K. Hur, and K. Yi, "Crellvm: Verified credible compilation for LLVM," PLDI '18, 2018.

[79] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 917–934, USENIX Association, 2017.