Protecting Cryptographic Libraries against Side-Channel and Code-Reuse Attacks

Rodothea Myrsini Tsoupidi, rtsoupidi@acm.org, *Independent Researcher**, *Stockholm, Sweden*
Elena Troubitsyna, elenatro@kth.se, *KTH Royal Institute of Technology, Stockholm, Sweden*
Panos Papadimitratos, ppapdim@kth.se, *KTH Royal Institute of Technology, Stockholm, Sweden*

Abstract—Cryptographic libraries, an essential part of cybersecurity, are shown to be susceptible to different types of attacks, including side-channel and memory-corruption attacks. In this article, we examine popular cryptographic libraries in terms of the security measures they implement, pinpoint security vulnerabilities, and suggest security improvements in their development process.

keywords: *cryptographic libraries, code-reuse attacks, side-channel attacks, compiler-based security*

Cryptographic algorithms and protocols constitute an essential part of software security. Cryptosystems, that is, ensembles of cryptographic algorithms, provide security services, such as confidentiality, authentication, integrity, and non-repudiation, based on mathematical problems deemed computationally intractable. Cryptographic libraries implement cryptographic algorithms in software and they offer an essential building block for implementing any cryptographic protocol. The importance of cryptographic libraries in security-critical applications has attracted malicious attackers that aim at exploiting the target system. For example, various cyberattacks [1, 2] have demonstrated the presence of exploitable vulnerabilities in different versions of OpenSSL[1], a popular cryptographic library, and other cryptographic libraries [3].

Side-channel attacks are powerful attacks that often target cryptographic software. These attacks allow unauthorized users to extract sensitive information during the program execution by recording different metrics, such as the execution time of the program. Cryptographic algorithms are specially targeted by side-channel attacks that aim at extracting secret information, in particular secret or private cryptographic keys. Revealing these values may allow an adversary to compromise a secure communication channel. For example, the adversary may access and/or modify at will end-to-end encrypted and authenticated messages, or impersonate a user using their private key to sign emails [1].

Another, highly exploitable class of software-enabled attacks is based on memory leaks. Memory-buffer vulnerabilities are among the most common security vulnerabilities, comprising approximately 20% of the reported vulnerabilities in cryptographic libraries [3]. Code-reuse attacks are among the most powerful attacks that are based on a memory corruption vulnerability, such as a buffer overflow, to alter the program control flow and hijack the target system. It is worth noting that many popular cryptographic libraries, such as OpenSSL, libsodium[2], and cryptlib[3], are implemented in unsafe programming languages, such as C and C++ [4], which allow memory-corruption vulnerabilities and unstructured control flow.

To deal with these threats, cryptographic-library developers implement security measures to protect the library implementation. These mitigations include source-code countermeasures and compiler-based mitigations. Note that timing side channel and code-reuse attacks require different types of countermeasures and applying these mitigations independently may compromise the security of the system [5].

A recent survey by Jancar et al. [4] studies mitigation techniques against timing side-channel attacks in popular cryptographic libraries. The study reveals that only 13 out of 27 libraries claim resistance against timing attacks. Moreover, the applied mitigation techniques in some libraries cannot guarantee the absence of timing side-channel vulnerabilities in the executable

*The main part of this article was written during the first author's work at KTH Royal Institute of Technology.

Digital Object Identifier 10.1109/XXX.0000.0000000

[1]OpenSSL: https://www.openssl.org/

[2]libsodium: https://nacl.cr.yp.to/

[3]cryptlib: https://www.cs.auckland.ac.nz/~pgut001/cryptlib/

code. The survey focuses on the developers' effort to protect their code and the use of verification tools in the testing process of the library to ensure the absence of timing leaks. Instead, in this work, we focus on the compilation process as the main mitigation stage.

Compilers play an essential role in the development of cryptographic libraries. In particular, cryptographic libraries typically use general-purpose optimizing compilers, such as GCC[4] and LLVM[5], to compile the source code and generate efficient code for a wide range of target processors. However, code optimization and compiler-based security are two conflicting goals [6], and general-purpose compilers often lean towards efficiency. Instead, secure compilation approaches allow for secure-by-design development of security-critical applications often at the cost of code efficiency.

In this work, we investigate the current mitigations of eleven popular cryptographic libraries against code-reuse attacks and side-channel attacks and discuss different existing compiler approaches that may improve security in cryptographic libraries. Many of these secure compilation approaches have drawbacks, such as supporting a subset of cyberattacks, introducing high overhead, or providing low maintenance. As a way to deal with these open challenges, we propose possible directions toward improving the compilation of popular cryptographic libraries:

- **General-purpose compiler developers** need to provide additional security countermeasures and control over the compilation procedure to the programmer;
- **Secure compilation approaches** need to target multiple attacker models, consider optimization goals, and improve maintenance;
- **Cryptographic library developers** need to adapt secure compilation approaches or enable additional protection options in their current compilation scheme.

In the following sections, we first describe how timing and code-reuse attacks operate and common mitigations against these attacks. Subsequently, we present a summary of compiler-based or post-compilation countermeasures in cryptographic libraries. Finally, we present a set of secure compilation approaches that may be used to protect cryptographic libraries and discuss possible directions for improving the security of cryptographic libraries.

[4]GCC: https://gcc.gnu.org/
[5]LLVM: https://llvm.org/

CYBERATTACKS

Cryptographic libraries are vulnerable to side-channel attacks. In particular, side-channel attacks measure time, power consumption, sound, or electromagnetic emissions, to infer security-critical values, notably cryptographic keys. Timing side-channel attacks are particularly effective against cryptographic implementations because they do not require physical access to the target device. For example, OpenSSL and PolarSSL (now Mbed TLS) implementations had a timing vulnerability, known as lucky thirteen vulnerability (CVE-2013-0169).

Furthermore, a cryptographic library is software that may contain memory vulnerabilities. More specifically, memory vulnerabilities threaten cryptographic implementations. For example, the Heartbleed vulnerability (CVE-2014-0160) in OpenSSL allowed reading sensitive memory. Code-reuse attacks constitute powerful attacks that allow full control of the system using a memory corruption vulnerabilities. In general, memory vulnerabilities may enable powerful code-reuse attacks in two ways: 1) initiate the attack using a memory corruption vulnerability that is present in the library and 2) use code snippets from the library code base to stitch a code-reuse attack.

Timing Side-Channel Attacks

Cryptographic implementations are typically written in high-level languages, such as C and C++. This code is translated to the target processor machine code, which consists of a set of hardware instructions. The execution sequence of these instructions and the values they process affect the observable execution time of the code. Precise time measurements allow the observer to extract information about the processed values. For example, the execution time of arithmetic division may take shorter time if the divisor is equal to one than otherwise.

Cryptographic libraries implement cryptographic protocols, which process security-sensitive values that should remain secret, such as an encryption key. The implementation of a cryptographic algorithm may lead to information leakage during execution. Figure 1a shows the implementation of function `check_bit`, which may be part of a naive implementation of authenticating a user-provided password (**`pub`**) compared to a stored password (**`key`**). Here, **`u8`** denotes an unsigned 8-bit value. Function `check_bit` takes two values as input, **`pub`** and **`key`**, and compares them. If the values are equal, the function returns one, otherwise, the function returns zero. We assume that one of the input arguments, **`key`**, is a *secret* value, e.g. part of the password, and the other value, **`pub`**, is known to a potential attacker. At line 3, the code compares the

```
1  u8 check_bit(u8 pub, u8 key) {
2    u8 t = 0;
3    if (pub == key) t = 1;
4    return t;
5  }
```

(a) Program with secret-dependent branching

```
1  int read_password(char *input)
2  {
3    char user_password[8];
4    strcpy(user_password, input);
5    ...
6    return 0;
7  }
```

(b) Program with buffer overflow

```
1  0x0008328c: ...
2  0x00083290: lw $ra, 0x24($sp)
3  0x00083294: lw $s0, 0x20($sp)
4  0x00083298: jr $ra
5  0x000832a0: addiu $sp, $sp, 0x28
```

(c) A code-reuse gadget in MIPS32

FIGURE 1: Cryptographic-library vulnerabilities

two input values and if they are equal, the program will assign the value of one to variable `t`. When the comparison is false, the code may take less time to run (assuming no branch calculation delays) because the processor does not need to perform the assignment at line 3. This means that depending on the value of **`key`**, the processor will take less or more time to execute the function. Therefore, an attacker that is able to measure the execution time of this function will be able to derive information about the value of **`key`**. Typical sources of timing leaks are secret-dependent branches (e.g. Figure 1a), secret-dependent memory addressing, and secret-dependent operands in variable-latency instructions (e.g. division).

Code-Reuse Attacks

Many cryptographic libraries (see Table 1) are implemented using unsafe languages with unstructured control flow, such as C and C++. The main advantages of these languages are high portability and increased efficiency. However, these languages allow the introduction of memory vulnerabilities that enable powerful attacks via control-flow hijacking, such as code-reuse attacks.

One direct source of code-reuse vulnerabilities in cryptographic libraries is the presence of memory corruption vulnerabilities in the library code. In particular, code-reuse attacks use a memory corruption vulnerability, such as a buffer overflow, to insert data into the program memory and alter the control flow of the program. The execution of the code may lead to arbitrarily complex attacker-controlled execution [7]. Memory corruption vulnerabilities, such as buffer overflows, appear when the program allows a user to write to unintended memory locations. For example, buffer-overflow vulnerabilities allow an attacker to overwrite parts of the memory beyond the limits of an array. Figure 1b shows function `read_password`, which reads a user-provided password to compare with a stored or hashed password. At line 3, the program defines `user_password`, an array of eight bytes. At line 4, the program copies the content of the user input in `input` to `user_password` using function `strcpy`. Neither the main program nor function `strcpy` checks whether `input` fits in `user_password`. Instead, `strcpy` copies the contents of `input` to the memory that starts at the address of `user_password`, potentially overwriting data that follow array `user_password`. Typically, compilers store locally defined arrays in the stack frame; thus, in Figure 1b, an input of more than eight characters results in overwriting the stack. The stack frame often contains the address to which the function will return to, and therefore, overwriting this address leads to a redirection of the control flow. An attacker may overwrite the return address of the function with an arbitrary address in the execution code to redirect the program execution to this arbitrary address.

The second source of code-reuse vulnerabilities in cryptographic libraries is the compiled code of the library. When the attacker identifies a memory corruption vulnerability in the cryptographic library or other parts of the executable program, the attacker inserts data that redirects the program execution and results in unwanted program behavior. The attacker payload consists of carefully selected code snippets, so-called gadgets, which when stitched together, result in an attack. In our context, the attacker may use the library code base as a pool for code-reuse gadgets that enable code-reuse attacks [2]. A gadget is a code snippet that typically ends with a control-flow instruction, which allows the attacker to direct the execution to the next gadget. Figure 1c shows a code-reuse gadget extracted from the GNU C Library (`libc`)[6] in a MIPS-based system using ROPGadget[7], a code-reuse gadget extraction tool. The gadget starts with a load operation, that loads an attacker-controlled value from the stack, `0x24($sp)`, to the return-address register, **`$ra`**. The second instruction (line 3) loads an attacker-

[6] libc: https://www.gnu.org/software/libc/
[7] ROPGadget: http://shell-storm.org/project/ROPgadget/

controlled value to register `$s0`. This value may be used by the attacker to perform a malicious operation, for example, to create a system shell. At line 4, the gadget jumps to the next attacker gadget, where the control flow continues to the address in register `$ra`. The last line[8] updates the stack pointer `$sp`, which allows the attacker to use a different part of the stack for the next gadget.

Code-reuse gadgets are present in almost every code implementation. Therefore, an executable and any dynamic library that is linked to this executable can provide the attacker with useful building blocks to construct an attack.

COUNTERMEASURES

This section describes countermeasures against timing side-channel attacks and code-reuse attacks, including both manual source-code mitigations and compiler-enabled countermeasures.

Timing Side-Channel Mitigations

Timing side-channel attacks are based on observations of secret-dependent timing variations during code execution. A well-known programming discipline to mitigate timing side channels is constant-time programming, where the execution time of the program should not depend on secret values. This means that the mitigated program does not contain any secret-dependent branch instructions, memory operations, or variable-latency instruction operands. Constant-time programming requires transforming secret-dependent code to constant-time equivalent. Often, developers implement this mitigation manually. Figure 2a shows a naive constant-time implementation of the code in Figure 1a [8]. The idea of this mitigation is that there is a single assignment of variable `t` regardless of the value of `key`. The compiler may translate the `?:` expression to constant-time code, such as `cmov` in x86 or `csel` in ARM, which is the desirable result. However, the compiler may translate the expression to an `if-else` statement, breaking the constant-time mitigation [9].

Figure 2b shows an alternative constant-time transformation of the code in Figure 1a. First, this implementation (line 3) creates a mask, `m`, which depends on the value of `key` and has value `0xFFFFFFFF` (if the values of `key` and `pub` are equal) or `0x00000000` (if the values are not equal). Then, the code copies the correct value, zero or one, using the previously generated mask. This transformation is more difficult for the compiler to revert. Nonetheless, different compiler versions and optimization flags may break this mitigation, by converting the code to a branch statement [9].

Constant-time implementations are often implemented manually and are, therefore, error-prone. Moreover, the code becomes difficult to understand and analyze. Figure 2c shows an alternative mitigation often referred to as constant-resource programming [5]. This mitigation adds dead code to ensure that both branch paths take the same time to execute (line 6). To ensure the correctness of this mitigation, the compiler needs to be aware and preserve this mitigation, otherwise, the compiler may remove the `else` branch because `_t` is an unused variable. In addition, this mitigation may require an accurate timing model of the target processor.

Code-Reuse Attack Mitigations

One direction towards mitigating code-reuse attacks is identifying and mitigating possible memory-corruption vulnerabilities in the code implementation. Identifying these vulnerabilities often includes static analysis of the code to discover possible vulnerabilities and/or add runtime buffer-overflow checks. A disadvantage is the inability to guarantee the absence of vulnerabilities in the program code and linked-library code, which may enable a code-reuse attack.

Stack canaries is a mitigation against buffer overflows on the stack. A canary is a randomly generated word at the end of the stack frame that is set at runtime. This word is checked before jumping out of the function to detect possible stack-smashing attempts. Apart from hindering illegal data insertion in the stack, stack canaries may also hinder control-flow violations in code-reuse attacks, such as Return-Oriented Programming (ROP) [7]. ROP attacks use code-reuse gadgets that end with a return instruction. Mitigating ROP attacks requires introducing stack canaries at the exit of every function. Unfortunately, there are ways around stack canaries, such as brute-force attacks, memory-disclosure attacks [10], or Jump-Oriented Programming attacks that use gadgets ending with non-return branches.

Automatic software diversification or randomization is a method to protect against most types of code-reuse attacks. The idea is that introducing randomness to the implementation of the code reduces the probability of a successful attack. For example, the gadget in Figure 1c is located at relative address `0x00083290` is `libc`. However, if we can relocate this gadget, the

[8]The MIPS architecture uses *delay slots*, which follow a branch instruction but are executed before the branch.

```
1  u8 check_bit(u8 pub, u8 key) {
2    u8 t;
3    t = (pub == key) ? 1 : 0;
4
5    return t;
6  }
```

(a) Naive constant-time transform

```
u8 t; i8 m;
m = -(pub == key);
t = (1&m) | (0&~m);
return t;
}
```

(b) Constant-time transform

```
u8 _t, t = 0;
if (pub == key) t = 1;
else _t = 1;
return t;
}
```

(c) Constant-resource transform

FIGURE 2: Transformations of the code in Figure 1a

TABLE 1: Cryptographic libraries with enabled compilation flags (CF) and post-compilation timing testing (PCT); SP stands for `-fstack-protector-strong`, SBS stands for `-param ssp-buffer-size`, PIE stands for `-fPIE`, FORT stands for the macro `-DFORTIFY_SOURCE`, and CTT stands for constant-time testing.

Library	CF				PCT [4]
	SP	SBS	PIE	FORT	CTT
BearSSL	✓				✗
Botan	✓*				✓
cryptlib	✓			2	-
Crypto++	✓				✗
GnuTLS	✓				✗
LibreSSL	✓				✗
Libgcrypt	✓				-
libsodium	✓*		✓		✗
Mbed TLS	✓				✓
OpenSSL	✓				✗
wolfTLS	✓	1			✗

* the flag is `-fstack-protector`

attacker payload will need to adjust the payload for this relocation. The most widely used diversification scheme against code-reuse attacks is Address Space Layout Randomization (ASLR). ASLR randomizes dynamically the base address of the executable and the addresses of the stack, heap, and dynamically loaded libraries. ASLR is an important mitigation against code-reuse attacks, however, brute-force attacks may recover the randomized base addresses. Fine-grained diversification approaches are typically more difficult to defeat and require more sophisticated attacks [2].

Another method to protect against code-reuse attacks is control-flow integrity [11]. Control-flow integrity approaches aim to ensure that the control flow of the program at runtime complies with the intended control flow. A main disadvantage of control-flow integrity schemes is high overhead and/or dependency on specialized hardware.

SECURITY IN CRYPTO LIBRARIES

Popular libraries use countermeasures to protect against timing side channels and code-reuse attacks. We selected eleven well-known cryptographic libraries that are open source, BearSSL[9] 0.6, Botan[10] 2.19.3, cryptlib 3.4.7, Crypto++[11] 8.8.0, GnuTLS[12] 3.6.16, LibreSSL[13] 3.8.0, Libgcrypt[14] 1.10.2, libsodium 1.10.18, Mbed TLS[15] 3.4.1, OpenSSL 3.1.2, and wolfTLS[16] 3.13.0. The first part of Table 1 shows the compilation flags in GCC or LLVM for the default build process of the library. The flags mainly activate buffer-overflow checks (FORT), stack canaries (SP and SBS), and full ASLR (PIE). In the table, ✓ denotes the presence of the flag in the compilation process, whereas an empty box corresponds to the absence of the flag. To extract these flags, we compile each library on an Intel®Core™i7-8550U processor at 1.80GHz with 16 GB of RAM running Ubuntu 18.04, using GCC version 7.5.0 and clang version 10.0.1. The second part of the table marks which of the libraries perform post-compilation timing testing during the development process, based on the results by Jancar et al. [4].

Timing Mitigations in Cryptographic Libraries: Jancar et al. [4] show that many developers are aware of timing attacks and possible mitigations. More specifically, many cryptographic libraries support constant-time cryptographic algorithms and/or re-implement known vulnerable algorithms, e.g. Advanced Encryption Standard, using the constant-time programming discipline. However, most cryptographic libraries are not able to provide guarantees that the compiled binary code is

[9] BearSSL: https://bearssl.org/
[10] Botan: https://botan.randombit.net/
[11] Crypto++: https://www.cryptopp.com/
[12] GnuTLS: https://www.gnutls.org/
[13] LibreSSL: https://www.libressl.org/
[14] Libgcrypt: https://gnupg.org/software/libgcrypt/
[15] Mbed TLS: https://www.trustedfirmware.org/projects/mbed-tls/
[16] wolfTLS: https://www.wolfssl.com/

constant time. Table 1 shows that only two libraries perform constant-time testing as part of their code development process [4]. These results indicate that most cryptographic libraries do not guarantee the absence of timing vulnerabilities.

In addition, Jancar et al. [4] mention the following problems: 1) the uncertainty in preserving security properties in popular compilers and 2) the inability of current compilers to incorporate the secret/public type system into their current system. These challenges require in-depth changes to the compiler infrastructure to take into account side-channel attacks and their countermeasures.

Code-Reuse Mitigations in Cryptographic Libraries: Several code-reuse attack mitigations are included as optional or default in GCC and LLVM.

A method to prevent a buffer overflow is to check the buffer size before starting to copy the data (see Figure 1b). Preprocessor flag `-DFORTIFY_SOURCE` in GCC and clang defines a macro that inserts code around possible buffer-overflow vulnerabilities, which performs runtime buffer-overflow checks. In particular, the macro replaces common vulnerable functions, such as `strcpy`, with wrapper functions that check whether the size of the destination buffer is large enough to store the source data. In Table 1 only cryptlib uses this flag during compilation.

In GCC and LLVM, flags `-fstack-protector` and `-fstack-protector-strong` add stack canaries to functions that may contain buffer overflows, namely functions that manipulate a buffer of at least `-param ssp-buffer-size` buffer size. All libraries in Table 1 use either of these flags with the default buffer size, eight bytes, apart from wolfTLS, which uses a smaller buffer size of one byte. Flag `-fstack-protector-all` provides stronger protection as it adds stack canaries to every function and can protect against ROP attacks. Unfortunately, none of the cryptographic libraries in Table 1 uses this flag.

ASLR is default in many operating systems, however, the address of the executable code is typically not randomized. In GCC and LLVM, flag `-fPIE`, enables ASLR for executables. This flag is only used by libsodium in Table 1.

In summary, Table 1 shows that all libraries implement stack canaries in selected functions, however, the majority of the libraries do not enable additional compiler mitigations against code-reuse attacks.

Cryptographic-Library Security Challenges: Based on the work by Jancar et al. [4] and our experiment, cryptographic-library developers appear to be aware of the importance of security mitigations in cryptographic code. In particular, developers use a set of mitigations against timing side channels and code-reuse attacks. For mitigating timing side channels, developers depend often on manual implementations, whereas for mitigating code-reuse threats, developers use the available compiler options.

Unfortunately, these mitigations are not enough to guarantee security; the compilation process may remove source-code mitigations, while the features and options that compilers provide do not guarantee the mitigation of advanced code-reuse attacks. For example, stack canaries (used by all libraries) are not enough to protect against different types of code-reuse attacks, such as JOP attacks. Furthermore, cryptographic-library developers do not perform binary-level testing to ensure that source-code mitigations remain in the compiler-generated code. One solution to these challenges is to incorporate security checks and mitigations during compilation.

SECURE COMPILATION

We present four secure compilation approaches integrated into conventional compilers, in particular, LLVM, a popular and freely available compiler infrastructure toolchain. We selected a set of diverse compiler approaches with regards to the method they use, however, this list is not complete. Table 2 shows information about these LLVM-compatible compilers, including the attacks they mitigate and the overhead they introduce (green denotes advantageous values, red denotes disadvantageous values). In the next sections, we describe each of these approaches.

Secure Compiler

The Vu et al. [12] approach targets multiple attacker models, including timing side channels, with the compiler based on a recent version of LLVM and evaluated for x86 systems and ARMv7-M/Thumb-2. Their paper introduces the notion of *opaque observations*, marking operations the compiler cannot remove or replace, while it can statically analyze them. *Opaque observations* are source-code annotations and are preserved throughout the compilation procedure, forcing the compiler to preserve constant-time source-code mitigations. To ensure that the code mitigation in Figure 2c is valid after compilation using the compiler by Vu et al. [12], the developer may write the following code:

TABLE 2: Compiler tools; LLVM lists the version of LLVM the tool is based on; TSC means that the tool provides mitigation(s) against timing side-channel attacks; CRA means that the tool provides mitigation(s) against code-reuse attacks; Avail lists whether the tool is publicly available; ETO_a stands for Execution-Time Overhead and CTO_a stands for Compilation-Time Overhead for $a \in \{TA, CR\}$.

Compiler	LLVM	TSC	CRA	Avail	ETO_{TA}	CTO_{TA}	ETO_{CR}	CTO_{CR}
SecComp [12]	∼11	✓	✗	✗	low	low	-	-
MCR [13]	3.8	✓	✓	✓	high	low	low	low
SecDivCon [5]	3.8	✓	✓	✓	low	high	low	high
PCFL [14]	13	✓	✗	✓	high	low	-	-

```
u8 check_bit(u8 pub, u8 key) {
  u8 _t, t = 0;
  if (pub == key) {
    t = 1;
  } else {
    _t = 1;
    // Property: observe(_t)
  }
  return t;
}
```

Code annotation, `//Property: observe(_t)`, ensures that the compiler will not remove the unused code in the `else` statement in Figure 2c. Vu et al. [12] provide improved speedup compared to unoptimized LLVM code, often considered secure, at a small compilation-time overhead.

Vu et al. [12] do not deal with code-reuse attacks. However, their approach may use the mitigations that LLVM provides, such as stack canaries. Unfortunately, these mitigations may be defeated by advanced attacks, such as ROP or JOP attacks. To reduce the effect of these attacks, a developer may combine Vu et al. [12] approach with additional mitigations against code-reuse attacks. However, this comes at a risk of breaking the security properties of the tool.

To summarize, the approach by Vu et al. [12] is a promising solution to the security/optimization gap and provides a solution that is compatible with the compilation procedures of modern cryptographic libraries. We believe that a similar solution can be adopted by compiler developers of GCC and LLVM to provide mitigations against more security threats with limited efficiency overhead. With regard to code-reuse attacks, Vu et al. [12] do not provide advanced mitigations, such as control-flow integrity or fine-grained randomization. An important hindrance to adopting this approach is that it requires some manual annotation, and more importantly, it is not freely available.

Multicompiler

Multicompiler (MCR) [13] is an open source[17], program randomization tool, based on LLVM 3.8, targeting x86 architectures. MCR supports different levels of software randomization with main focus on mitigating code-reuse attacks [13]. MCR takes a random seed as a command-line argument and generates diverse code given different random seeds. The tool does not require any additional annotations.

MCR compiles a C program using a modified version of LLVM. The tool supports stack layout randomization using flag `-mllvm -shuffle-stack-frames`, function randomization with `-mllvm -randomize-function-list`, no-operation insertion with `-Xclang -nop-insertion`, hardware register randomization with `-mllvm -randomize-machine-registers` and more. To generate diverse programs in every compilation, MCR uses `-frandom-seed=<seed>`. For example, to compile a file name password.c, you can write:

```
$ clang password.c -frandom-seed=123 \
  -Xclang -nop-insertion             \
  -mllvm -randomize-machine-registers\
  -o password
```

The idea of compiler-based randomization is that different users generate and run different program variants, which complicates potential code-reuse attacks. MCR may also be used against timing side-channel attacks by inserting random memory `load` instructions, however, in this context, the tool introduces a very high execution-time overhead of up to eight times slowdown [15].

The compilation time of the tool is comparable with LLVM. Similarly, the execution-time overhead is typically small (and controllable), up to 25% for no-operation insertion [13], however, the execution-time

[17] MCR: https://github.com/securesystemslab/multicompiler.git

overhead is high when mitigating timing side-channel attacks [15].

In summary, MCR is appropriate for defending against code-reuse attacks, but it is not highly efficient as a mitigation of timing side-channel attacks. An additional drawback is that it is based on a relatively old version of LLVM (see Table 2). However, the tool is available online, has small compilation time, and supports whole-program compilation.

SecDivCon

Secure-by-construction Code Diversification (SecDivCon) [5] is a constraint-based compiler that allows the generation of efficient and secure code. SecDivCon is open source[18]. It is based on Unison, a combinatorial compiler backend[19]. The tool is compatible with LLVM version 3.8 and targets a generic MIPS processor and ARM Cortex M0.

SecDivCon defines compiler-backend transformations and security requirements as a set of constraints. SecDivCon uses a constraint solver to find diversified code implementations that are highly optimized with regard to execution time and secure against timing side channels. The tool enables constant-resource mitigations and backend diversification, such as no-operation insertion, hardware register diversification, instruction shuffling, and register and memory copy insertion.

To mitigate timing side channels, SecDivCon requires defining the security policy, namely which variables are security sensitive (Secret) and which variables do not leak any information (Public). For example, to ensure that the code in Figure 1a is constant resource, the user need to define the following policy:

```
Public a0;
Secret a1
```

Notations `a0` and `a1` correspond to the argument registers that contain the input arguments. Given the security policy, SecDivCon generates automatically diverse programs that are secure against timing side channels.

SecDivCon has several advantages: 1) combines performance and security goals, 2) guarantees the preservation of properties in the generated code, and 3) does not require extensive annotations. These advantages come at the cost of compilation complexity, which makes SecDivCon practical for small, vulnerable cryptographic functions. Thus, SecDivCon needs to be combined with different compilation approaches to compile large code bases.

In summary, SecDivCon offers strong guarantees for mitigating timing side channels for predictable devices, and provides a fine-grained diversification scheme, with high gadget diversification ability. The main disadvantage of SecDivCon is the high compilation time, the lack of support for advanced computer architectures, and the dependence on a relatively old version of LLVM.

Partial Control-Flow Linearization (PCFL)

Partial Control-Flow Linearization (PCFL) [14] is a linearization tool that converts a non-constant-time program to constant time. PCFL is available online[20] and uses LLVM 13.

The tool requires annotation of secret and non-secret values and is able to generate constant-time programs automatically without requiring manual effort from the developer. The following code snippet, shows how to use PCFL to mitigate the function `check_bit` in Figure 1a.

```
int main() {
  u8 in[4];
  secret u8 sec[4] = {0, 0, 0, 0};
  for (int i=0; i<4; i++)
    check_bit(in[i], sec[i]);
  return 0;
}
```

Notation **`secret`** ensures that the compiler generates constant-time code for the annotated memory locations. Note that PCFL is able to handle more complex programs, for example, **`for`** loops.

Table 2 shows a high execution-time overhead for the transformed code by PCFL, however, the comparison considers non constant-time code, which is not the case with the work by Vu et al. [12], where the source code is already constant time. PCFL focuses on timing side channels and is able to provide guarantees in the generated code. This approach has several advantages including, automatic constant-time code generation, low compilation overhead, and high program coverage.

Similar to the work by Vu et al. [12], PCFL does not support mitigations against code-reuse attacks apart from the LLVM optional compiler mitigations. Performing advanced post-compilation mitigations would require reassessing the mitigation guarantees by PCFL. In summary, PCFL is a potential solution for reducing the overhead and final-code uncertainty of manual constant-time implementations.

[18]SecDivCon: https://github.com/romits800/secdivcon_experiments.git

[19]Unison: https://github.com/unison-code/unison

[20]PCFL: https://github.com/lac-dcc/lif

OPEN CHALLENGES

The different compilation approaches we discussed in the previous section provide possible solutions for compiling cryptographic libraries. These tools are based on LLVM, which is mostly compatible with the current compilation schemes of the cryptographic libraries in Table 2.

The work by Vu et al. [12] is among the most appropriate approaches for dealing with timing side-channels because it provides control over the compilation process and supports constant-time programming. Unfortunately, the tool is not publicly available. PCFL may be used in cryptographic libraries that do not provide any constant-time source-code mitigations to protect potentially vulnerable parts of the code automatically. Similarly, SecDivCon may provide a method to mitigate timing attacks in insecure source code for embedded devices. SecDivCon may also be used in a diversification scheme to harden cryptographic libraries against code-reuse attacks. MCR provides whole-program diversification and may be used against both code-reuse attacks and to hide timing vulnerabilities in libraries that do not implement source-code timing mitigations. However, MCR introduces significant overhead when used against timing side channels, which may be an important hinder in adapting MCR in performance-sensitive applications. To summarize, depending on the focus of a cryptographic library and the current implementation, different compilation approaches may be valuable to enhance the security of the library.

Each of the proposed secure compilation approaches has limitations: MCR only focuses on diversification, SecDivCon can compile only small functions, the tool by Vu et al. [12] and PCFL do not target code-reuse attacks, while the former is not available online. In addition, none of the presented tools targets RISC-V, a popular instruction-set architecture that is supported by LLVM.

We believe that bridging the optimization/security gap requires additional effort:

- **General-purpose compiler developers:** need to provide more control over the compiler outcome, allowing control over optimizations.
- **General-purpose compiler developers:** need to implement infrastructure and further security mitigations, including constant-time preservation.
- **Compiler developers:** need to focus on generating optimized code by redesigning compiler optimizations to be security aware.
- **Secure-compiler developers:** need to design tools that combine multiple mitigations instead of focusing on a single mitigation.
- **Cryptographic-library developers:** should make sure to implement constant-time alternatives and enable security flags in the compiler, which we have seen to some extent in this article.

CONCLUSION

This article describes security mitigations in popular cryptographic libraries and proposes the use of secure compilation approaches to enhance their security. More specifically, most popular libraries depend on manual mitigations and general-purpose compilers that provide secure solutions to a number of vulnerabilities. Unfortunately, general-purpose compiler mitigations are not sufficient against advanced attacks and do not always preserve source-code mitigations. Improving the security of cryptographic libraries requires changes in 1) the current development process of cryptographic libraries to consider using secure compilers, 2) the compilation approaches to allow for additional mitigations and more transparency, and 3) the secure-compilation approaches to combine multiple mitigations in one tool.

ACKNOWLEDGMENTS

The work of P. Papadimitratos is supported in part by the Knut and Alice Wallenberg Academy Fellow Trustworthy IoT project. E. Troubitsyna is partially supported by Swedish Foundation for Strategic Research with project FUS21-0026 *SUCCESS: Sustainable Cyber-Physical Software-Defined System Slicing*.

REFERENCES

1. Y. Yarom, D. Genkin, and N. Heninger, “Cachebleed: a timing attack on openssl constant-time rsa,” *Journal of Cryptographic Engineering*, vol. 7, pp. 99–112, 2017.
2. S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. D. Yao, “Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20, Oct. 2020, pp. 1803–1820.
3. J. Blessing, M. A. Specter, and D. J. Weitzner, “You really shouldn’t roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries,” *arXiv preprint arXiv:2107.04940*, 2021.
4. J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, “"They’re not that hard to mitigate": What crypto-

graphic library developers think about timing attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 632–649.
5. R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," *Computers & Security*, vol. 133, p. 103405, 2023.
6. V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.
7. H. 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*, ser. CCS '07, 2007, pp. 552–561.
8. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying {Constant-Time} implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.
9. L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1021–1038.
10. H. Marco-Gisbert and I. Ripoll-Ripoll, "SSPFA: effective stack smashing protection for android os," *International Journal of Information Security*, vol. 18, no. 4, pp. 519–532, 2019.
11. N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.
12. S. T. Vu, A. Cohen, A. De Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.
13. A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided Automated Software Diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. IEEE Computer Society, 2013, pp. 1–11.
14. L. Soares, M. Canesche, and F. M. Q. Pereira, "Side-channel elimination via partial control-flow linearization," *ACM Transactions on Programming Languages and Systems*, 2023.
15. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity," in *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

Rodothea Myrsini Tsoupidi is a independent Researcher in Stockholm, Sweden. Her research interests include compiler optimization, software diversification, language-based security, and side-channel mitigations. She received a Ph.D on Information and Communication Technology from the school of Electrical Engineering and Computer Science, at Royal Institute of Technology KTH, Sweden.

Elena Troubitsyna is a Professor at KTH – Royal Institute of Technology, Department of Computer Science, where she leads a research group that studies techniques for the development of dependeable-by-construction systems. Her research focuses on creating formal and model-driven methods for the development of safety- and security-critical software-intensive systems. Elena Troubitsyna received her Ph.D. Degree from the Turku Center for Computer Science.

Panos Papadimitratos leads the Networked Systems Security group and he is a member of the steering committee of the Security Link center, KTH. Papadimitratos received his Ph.D. degree in Electrical and Computer Engineering from Cornell University. He is a fellow of the Young Academy of Europe, a Knut and Alice Wallenberg Academy Fellow, an IEEE Fellow, and an ACM Distinguished Member. He serves or served as: member (and currently chair) of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) steering committee; member of the Privacy Enhancing Technologies Symposium (PETS) Editorial and Advisory Boards and the International Conference on Cryptology and Network Security (CANS) Steering Committee; program chair for the ACM WiSec 2016, International Conference on Trust & Trustworthy Computing (TRUST) 2016 and CANS 2018 conferences; general chair for ACM WiSec 2018, PETS 2019 and IEEE European Symposium on Security and Privacy 2019 conferences; and Associate Editor of the IEEE Transactions on Mobile Computing, ACM/IEEE Transactions on Networking, Institute of Engineering and Technology Information Security and ACM Mobile Computing and Communications Review journals. His group webpage is: https://www.eecs.kth.se/nss.