Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK

Xuancheng Jin\textsuperscript{1}, Xuangan Xiao\textsuperscript{1}, Songlin Jia\textsuperscript{1}, Wang Gao\textsuperscript{1}, Dawu Gu\textsuperscript{1}, Hang Zhang\textsuperscript{2}
Siqi Ma\textsuperscript{3}, Zhiyun Qian\textsuperscript{2}, and Juanru Li\textsuperscript{1}

\textsuperscript{1} Shanghai Jiao Tong University \textsuperscript{2} UC Riverside \textsuperscript{3} The University of Queensland

Abstract—Protecting confidential data against memory disclosure attacks is crucial to many critical applications, especially those involve cryptographic operations. However, it is neither easy to identify involved cryptographic confidential data in a program nor to implement a fine-grained and yet efficient protection. Existing defensive techniques face many shortcomings such as coarse-grained protection or exorbitant overhead. As a result, real world crypto applications seldom applied this kind of protection in practice.

To make the protection of cryptographic confidential data practical, we design and implement CryptoMPK, a source code analysis and transformation system to implement a domain-based memory isolation. CryptoMPK first automatically tracks and labels all sensitive memory buffers and operations in source code with a context-sensitive, crypto-aware information flow analysis. Then it partitions the source code into crypto and non-crypto domains with a context-dependent privilege switch instrumentation. By further utilizing Intel Memory Protection Keys (MPK), CryptoMPK generates executables with efficient domain switching, protecting them against typical memory disclosure vulnerabilities such as arbitrary memory read. In particular, by using CryptoMPK, a large number of intermediate memory buffers that have been previously ignored before are well protected, and thus the security risks are reduced significantly. We leveraged CryptoMPK to protect prevalent applications such as Apache and Nginx with widely used crypto libraries (e.g., OpenSSL, LibSodium). CryptoMPK only needs several minutes to analyze each of these complex cryptographic programs and incurs at most 9.53\% performance overhead for the protected programs.

I. INTRODUCTION

Applications rely on the confidentiality of crypto keys to guarantee the security of executed crypto operations (e.g., HTTPS encryption). Hence, memory disclosure attacks, which could leak the secret crypto key, severely threat the security of those applications. Allowing every instruction of a process to access crypto data is dangerous and unnecessary. In response, various solutions have been proposed in recent years to enhance the protection of crypto-related confidential data in process memory.

A natural strategy is to introduce in-process memory isolation to a program to add special access control for confidential memory buffers. However, doing this correctly and securely can be tricky for crypto software since it is difficult to identify all relevant data that need isolation manually. One of the basic principles of modern cryptography, the Kerckhoffs’ Principle [1], states that everything about the cryptosystem, except the key, is public knowledge. This principle, however, misleads many memory protection solutions: Most existing approaches only protect the main crypto keys only, which is insufficient to defend against memory disclosure attacks. For instance, some approaches (e.g., Safekeeping [2], Copker [3]) keep crypto keys solely inside CPU, without leaking their plaintext to memory. Unfortunately, most software implementations of modern ciphers (e.g., AES, RSA) generate and store intermediate states of the crypto operations in memory as well, from which the crypto key can be easily derived. Thus the memory disclosure attacks still work even when the proposed defenses are deployed.

Aside the crypto key, a wide variety of runtime data (e.g., the CryptReleaseContext of Windows Crypto API) also contain confidential information that needs protection but frequently ignored by most existing defense mechanisms. However, protecting all crypto confidential data in a program is a very sophisticated and challenging task, especially for those real-world complex programs and crypto libraries. It needs to first identify all memory buffers containing confidential data (i.e., crypto buffers), and then memory operations allowed to access them (i.e., crypto operations). Most protection approaches identify crypto buffers and operations manually (e.g., xMP [4] and libmpk [5]), which is time-consuming and error-prone even for an expert developer. Some other works (e.g., DataShield [6], SeCage [7]) leverage program analysis to automatically locate crypto buffers, however, due to the unawareness of the intrinsic nature of data propagation of crypto operations, they have a severe over-tainting issue (e.g., encrypted ciphertext or decrypted plaintext are also treated as confidential). By allowing more code to access those over-tainted data, attack surfaces to disclose secrets expand significantly. Another problem is how to isolate crypto and non-crypto operations in a process while minimizing the performance overhead. Some defenses (e.g., ConfLLVM [8]) adopt an address-base isolation by inserting checks before every memory access instruction. Although the protection is fine-grained, for most crypto operations that are CPU and memory access intensive, this is obviously inefficient and unacceptable.

Major challenges for generating a crypto secrets protected program include how to track and label all sensitive data and operations in the code automatically, how to grant the permission of an instruction to access sensitive data, and how to implement and deploy a fine-grained yet efficient
isolation. In this work, we overcome the above challenges by developing a comprehensive in-process isolation solution specifically designed to protect sensitive crypto-related secrets. At a high level, our solution, CRYPTO MPK, relies on a fine-grained and crypto-aware static taint analysis whose results inform a subsequent partition of the program into different protection domains. Specifically, CRYPTO MPK is equipped with three core techniques to address the aforementioned challenges: 1) CRYPTO MPK automatically determines what data to protect given a small number of initially labelled tags. In particular, CRYPTO MPK avoids the over-tainting issue by conducting a crypto-aware ciphertext/plaintext declassification. 2) CRYPTO MPK adopts a context-sensitive instrumentation to partition the program in a very precise way. Unlike traditional access control schemes, the attributes of an instruction (e.g., whether it needs to be isolated from a certain piece of sensitive information) can change under different calling contexts according to our analysis results, enabling a more accurate and efficient protection. CRYPTO MPK achieves such a context-sensitivity by generating multiple copies of code instrumented with context-dependent privilege switches. 3) CRYPTO MPK enforces an efficient domain-based isolation by using the Memory Protection Keys (MPK) hardware feature of latest Intel processors. Furthermore, CRYPTO MPK optimizes the protection granularity to minimize the runtime overhead while maintaining strong security guarantees.

We evaluated CRYPTO MPK on four server programs (Apache, Nginx, OpenSMTPD, vsftpd) with five widely used open source crypto projects (OpenSSL, libodium, libhydrogen, ccrypt, glibc-crypt) to examine its effectiveness and efficiency against real-world crypto scenarios including file encryption, message protection, SSL/TLS transmission, and password authentication. The results show that CRYPTO MPK can accurately identify both crypto buffers and crypto operations, and provide reliable protection for both crypto keys and other crypto-related sensitive information without affecting the normal software functionalities, significantly reducing the exposed confidential data. Regarding the performance, it took only several minutes for CRYPTO MPK to analyze large and complex projects offline, and the runtime overhead is at most 9.53% for real-world crypto applications.

In summary, we make the following contributions:

1) We have designed and implemented the CRYPTO MPK system to automate the tasks of confidential crypto data/code identification (including intermediate data derived from the original) and protected executable generation that incurs negligible overhead at runtime.

2) We show that secret information leaking through intermediate buffers (derived from the original secret) is very common but often ignored for most crypto applications, leading to broken security guarantees.

3) We applied CRYPTO MPK to state-of-the-art crypto libraries such as OpenSSL [9] and large server programs such as Apache [10] against any memory disclosure attacks (including the ones that attempt to read the intermediate buffers) to demonstrate the effectiveness of our solution.

4) We developed an empirical guideline to fully utilize MPK to protect crypto programs in a practical manner. Our solution could be helpful for the protection of other types of sensitive data in computational-intensive code.

**Availability.** We have open sourced our solution at https://cryptompk.code-analysis.org, where we placed the tool chain, instructions of our experiments, and the tested programs (source code and pre-built binaries).

II. BACKGROUND

A. Problems and Requirements

By transforming memory corruption vulnerabilities into arbitrary memory read primitives, attackers can disclose sensitive data such as crypto keys. In response, many defenses aim to protect credential information in memory. The problem is that for a cryptosystem, if not all sensitive data are protected, the defense would be ineffective. We use an example here to illustrate the difficulties in protecting crypto buffers where secrets are stored in crypto operations. Listing 1 shows the abstracted AES encryption in ccrypt [11], a cryptographer developed data encryption tool: In this example, the encrypt function generates an AES key from a secret passwd. The key generation procedure propagates the secret information of passwd to several intermediate buffers such as the rkk buffer and the keyblock buffer. Consider the situation that an attacker leverages an arbitrary memory read vulnerability to scan the memory, not only the passwd buffer but all those intermediate buffers that store crypto secrets should be protected. Otherwise, attacker could still exploit the leaked intermediate buffers to recover secret keys.

```
void encrypt(void *ciphertext, void *plaintext, 
             char *passwd) {
    roundkey rkk;
    hashstring(passwd, ...
    xrijndaelEncrypt(ciphertext, plaintext, rkk);
}
```

Listing 1: An example of secret propagation in ccrypt

Unfortunately, existing works often fail to comprehensively protect crypto sensitive information. Many protection approaches intuitively focus on an original secret but ignore its propagation. Take the case of using libmpk [5] to protect OpenSSL against the infamous HeartBleed vulnerability [12] as an example. The protection only defended SSL/TLS private keys from potential information leakage by storing the keys in isolated memory pages. In particular, they only protected data types that store private keys (e.g., EVP_PKEY) with an isolated memory region, but do not consider the secret propagation and let secret information leak to intermediate buffers. Although they additionally protected functions that need to access the private key, we will demonstrate in Section V-E that if only the private key is protected, an attacker could exploit HeartBleed...
vulnerability to retrieve other intermediate states and still recover the secret key.

B. Challenges

In short, the major research challenge of this paper is to determine the minimum part of code that we must grant crypto buffer access privilege, and block any other accesses (e.g., a buffer overread somewhere in the application process). To accurately label all crypto buffers in a process, and implement a fine-grained in-process isolation to restrict irrelevant code from accessing secret information. Existing works face the following challenges:

**How to accurately label crypto memory buffers and related operations?** Crypto data are widely distributed at different regions of process memory (e.g., stack, heap, global variable). Existing solutions (e.g., DataShield [6], MemSentry [13], libmpk [5] and xMP [4]) provide primitives to isolate sensitive information in specific memory regions. Nonetheless, they do not answer the question of how to find all those crypto memory buffers automatically. A straightforward idea (e.g., the solution adopted by SeCage [7]) to find all crypto buffers is to track the propagation of initial secret with taint analysis, and restrict the access against all reference data. This strategy, unfortunately, suffers from over-protection issue because it propagates taint tag to plaintext/ciphertext. In the example of Listing 1, if we define the password as the taint source, a standard taint analysis would also taint the ciphertext at Line 6. In this situation, the tainting of ciphertext buffer (and its following propagation) will significantly enlarge the protection scope.

**How to isolate crypto data from unauthorized memory accesses appropriately?** The isolating of crypto data inherently incurs performance overhead. A very fine-grained isolation may introduce unacceptable overhead, especially for CPU-intensive and time-sensitive crypto operations. On the other hand, a less fine-grained isolation may leave an attack more opportunities in stealing sensitive data. Therefore, it is important to choose an appropriate protection granularity to balance security and performance overhead.

**How to implement the isolation without changing the execution model?** Some in-process isolation approaches require to modify the execution model of the program (e.g., moving part of the program to TEE or SGX). This involves heavy modification of existing code and often introduces compatibility issue, thus is often infeasible in most cases.

C. Solutions

**Analysis-driven Sensitive Data Labeling (Section III-C).** Instead of asking developers to manually label crypto memory buffers and related operations, we make use of a static code analysis to help label those information automatically. Particularly, the adopted static code analysis employs a special crypto-aware, context-sensitive information flow propagation strategy to label crypto memory buffers effectively.

**Hardware-supported Domain Isolation (Section III-D).**

To protect labeled sensitive data, we associate them with a crypto domain, and fulfill a domain based isolation, which adds an access permission to qualify the access of crypto secret. When the execution shifts between crypto and non-crypto domains, instead of using software based heavyweight access permission granting mechanisms (e.g., hypervisor based), we utilize Intel Memory Protection Keys (MPK) technology [14] to implement a processor-featured privilege switching. This guarantees an efficient protection against crypto applications.

**Compiler-assisted Code Transformation (Section III-E).**

Instead of partitioning the program into different modules to implement isolation, we leverage the LLVM infrastructure to conduct an IR level code transformation (i.e., inserting privilege switching instructions, re-allocating memory buffers, adjusting call graph, etc.) while keeping the original execution model. Without porting the entire crypto application or any part of it to a secure world (e.g., SGX or Trustzone), a protected executable still runs in the same environment without special deployment.

III. CRYPTO

A. Threat Model and Concepts

In this paper we assume attackers are able to launch memory disclosure attacks due to memory safety issues. The goal of an attacker is to disclose crypto secrets (e.g., AES key, RSA private key) from the memory of a process. We only focus on memory disclosures caused by vulnerabilities such as HeartBleed [15] and do not consider attacks that compromise the integrity of execution (e.g., a control flow hijacking caused by an arbitrary memory write). We do consider the solutions that protect the integrity of a program [16], [17], [18] can be complementary to ours. In fact, as we show later in §V-D, we have partially integrated with one such solution successfully. Nevertheless, ensuring perfect integrity is still an unsolved problem which we currently consider to be out-of-scope.

We assume code vulnerabilities exist in an application, and attackers could remotely exploit such vulnerabilities to access sensitive data (e.g., crypto keys) in process memory of the vulnerable application. We do not limit the typ of memory disclosures, which could either be an out-of-bound read or an arbitrary address read. Nevertheless, this paper only focuses on memory safety issues and does not consider information leakage related side-channel attacks, both physical (e.g., cold boot attack [19]) and logical (e.g., Meltdown [20] and Spectre [21]) ones.

We use the concepts of crypto buffer and crypto operation to help analyze the program. We define them as follows:

**Definition 1:** A crypto buffer is defined as a continuous memory region (stack, heap, or global) that stores crypto secrets including master keys and intermediate results of cipher executions. Note that a buffer stores plaintext or ciphertext is not considered as a crypto buffer because it does not need to be protected.

**Definition 2:** A crypto operation is a memory operation (either read or write) that access a crypto buffer. We consider crypto operation to be security-sensitive if it accesses crypto buffer, and thus should be protected to guarantee the confiden-
tiality. For example, a memory read instruction that accesses
round key of AES encryption is defined as a crypto operation.

B. Overview of CRYPTO

Our solution, CRYPTO, is a source code analysis and
transformation system that adds fine-grained, context-sensitive
in-process isolation for credential data in crypto applications.
At a high level, CRYPTO labels crypto buffers and crypto
operations in a program automatically, and isolates crypto and
non-crypto operations to reduce the unnecessary accesses of
credential data. As Figure 1 depicts, CRYPTO adopts a
workflow consisting of a pre-analysis phase and three major
phases. In its pre-analysis phase, CRYPTO compiles
program source code to LLVM Intermediate representation
(IR) for the following analyses, and a manual annotation
on source code is required to help CRYPTO understand
specific crypto attributes of the analyzed target. After the pre-
analysis phase, CRYPTO first conducts a crypto buffer
labeling, which relies on static points-to analysis and taint
analysis to label crypto buffers. Next, CRYPTO employs
a crypto operation identification to locate memory accesses
and management operations on crypto buffers. After that,
CRYPTO leverages the results of previous analyses to
fulfill a code transformation. The transformation relocates la-
cled crypto buffers into a protected memory region, replicates
functions to be protected for different contexts, and partitions
the program into a crypto domain and a non-crypto domain,
instrumenting privilege switches on crossing boundaries. Fi-
nally, CRYPTO generates crypto domain isolated binary
code, and utilizes supplementary security checks to inspect
and eliminate potential bugs (e.g., using binary code rewriting
to remove ROP gadgets from the executable).

To identify crypto buffers and operations, CRYPTO
traverses source code of an analyzed program and handles each
instruction using a crypto-aware and context-sensitive taint
propagation. The crypto-aware feature proves that CRY-
ROTO can avoid over-labeling buffers such as ciphertext, and
the context-sensitive feature guarantees a context-dependent
domain isolation. Based on the identification results, CRY-
ROTO modifies LLVM compiler to generate programs with
customized memory management and in-process privilege
switching, which implement a crypto domain based isola-
tion. Only instructions in the crypto domain are allowed to
access crypto buffers, and any sensitive data access from non-
crypto domain will trigger a segmentation fault, which defends
against unauthorized memory accesses such as memory dis-

C. Automated Crypto Buffer Labeling

CRYPTO employs an automated crypto buffer labeling
to replace the traditional time-consuming and error-prone
manual annotation. It only requires developers to manually
annotate which buffer stores the initial secrets, and then uses
the initially annotated secret data as its taint source. For
instance, the encryption key and the private key are initial
secrets for symmetric-key ciphers and asymmetric-key ciphers,
respectively. With the annotated initial secrets as taint sources
(with a crypto taint tag), CRYPTO propagates taint tag
to label crypto buffers. It starts with the direct usages of
initial secrets, and traverses the IR instructions as abstractly
interpreting the code. It also leverages a points-to analysis
to accompany the taint analysis, determining the target of
pointers, improving the accuracy of both control-flow and data-
flow construction. After the traversal, CRYPTO generates
an abstract representation for each variable, which contains
taint tags and points-to information.

Unlike normal taint analysis, the taint propagation of CRY-
ROTO is crypto-aware. It considers the declassification of
plaintext/ciphertext. That is, unlike common taint analysis that
propagates the crypto tag to the output ciphertext or plaintext
(and thus leads to an over-tainting from the perspective of
cryptography), our crypto-aware taint analysis adds the tag of
crypto key to neither plaintext nor ciphertext. To handle this
case, our analysis requires an extra manual annotation against
plaintext and ciphertext buffers. Then, in the crypto-aware taint
propagation, CRYPTO assigns those annotated buffers
with a special mutually-exclusive (mxor) tag. When the
cipher mixes crypto keys with either plaintext or ciphertext,
the mxor tag will “eliminate” the crypto tag. This helps
CRYPTO fulfill the declassification of plaintext/ciphertext
without specifically defining their propagation rules.

Moreover, CRYPTO uses context-sensitive analysis to
attribute different tags for a same memory buffer under multi-
ple contexts. Taking Listing 2 as an example, the xreadline
function (Line 1-8) is used to read a text line from a file, and
allocate a buffer to store the read content. In this example, the
property of memory buffer allocated by xreadline (Line 2)
depends on the invoking context. If xreadline is invoked by the
readkey function (Line 12), the allocated memory buffer
should be labeled. In other cases (e.g., invoked by the prompt
function at Line 19) the allocated buffer is not sensitive.
In response, CRYPTO generates multiple contexts for
each function according to the execution. In each context, the
abstract representation of involved variables are independent.
Hence CryptoMPK could assigns more than one property for a variable. With such a context-sensitive analysis, CryptoMPK executes a fine-grained isolation in the following code transformation phase (detailed in Section III-E2).

D. Crypto Operation Identification

After labeling crypto buffers, in its second phase, CryptoMPK identifies which parts of the program are allowed to access those crypto buffers. At the IR level, CryptoMPK defines crypto operations as instructions that access memory of crypto buffers. Since the first phase analysis has already obtained the abstract representation of each variable, in a second traversal against the IR, CryptoMPK leverages the obtained information to identify crypto operations. In particular, CryptoMPK identifies two types of crypto operations.

1) Memory Access: In LLVM IR, load and store are used to access memory. By checking the accessed address arguments (as crypto or non-crypto buffer) of these instructions, CryptoMPK determines which instructions access crypto buffers, and attributes them as crypto operations.

2) Memory Management: To further manage crypto buffers, CryptoMPK introduces its customized memory management to replace the existing ones in a program. Thus it also needs to identify memory allocation and de-allocation operations. CryptoMPK considers a memory allocation operation as sensitive if it allocates a crypto buffer, and a memory de-allocation operation as sensitive if its pointer argument points to a crypto buffer.

E. Code Transformation

With the labeled crypto buffers and the identified crypto operations, CryptoMPK performs a code transformation at LLVM IR instruction level to partition the program into a crypto and a non-crypto domain. It first relocates crypto buffers to a protected memory region, and then inserts privilege switches between crypto and non-crypto domains.

1) Memory Relocation: CryptoMPK relocates different kinds of crypto buffers into a protected memory region in a unified way. In detail, CryptoMPK replaces both stack and heap allocations with functions of a customized memory manager (see details in Section IV-F). Next, CryptoMPK replaces the corresponding de-allocation functions for on-heap buffers with the customized one for protected buffers, and inserts extra de-allocation operations in function epilogues to explicitly release protected buffers that are originally on stack. For global crypto buffers, CryptoMPK re-allocates them during the initialization of the program and de-allocates them at the program termination.

Since the labeling of crypto buffer is context-sensitive, CryptoMPK correspondingly executes a context-dependent memory allocation. In Listing 2, for example, the xreadline function allocates buf for data read from external files. However, for different contexts, the allocated memory buffer could be either a crypto buffer or a non-crypto one. In this example, only the allocated buffer under the context of readkey (Line 12) is labeled as crypto buffer. Thus CryptoMPK does not directly replace malloc in Line 2. Instead, it replicates the xreadline function (one for crypto buffer allocation, and the other for non-crypto buffer allocation) to implement a fine-grained management (see Section IV-F).

Instead of invoking time-consuming system calls (e.g., mprotect) to implement a memory page level access control, CryptoMPK leverages the Intel MPK instructions (e.g., wrpkru) to fulfill an efficient privilege switch. MPK features a protection key (PKEY, stored in the 32-bit PKRU register) to control the access permission of a memory page. By binding a PKEY to a group of memory pages, access permission of those pages can be set simultaneously. CryptoMPK first relocates those labeled crypto buffers to a memory region maintained by its own memory manager, and then binds memory pages in this protected region with a PKEY.

2) Privilege Switch Instrumenting: To grant or revoke access permission against protected buffers at runtime, CryptoMPK instruments privilege switches between crypto domain and non-crypto domain. User process executes a wrpkru instruction to updates the PKRU register. CryptoMPK thus adds crypto domain prologue and crypto domain epilogue to the binary code to fulfill MPK privilege switching as Listing 2 illustrates. Only after an execution of crypto domain prologue, the later crypto operations are allowed to access protected memory buffers, and after an execution of crypto domain epilogue, protected memory buffers restore to an inaccessible state for non-crypto operations.

For current MPK-enabled processors, up to 16 PKEYs can be used simultaneously. This allows CryptoMPK to support more than one crypto domain with multiple crypto keys. In this paper, we only considered the protection with one crypto domain. Also note that the privilege switch mechanism is independent to other parts of CryptoMPK. Hence, it could be easily extended to adapt different scenarios. For instance, in July 2020, Intel have proposed a new PKS [22] feature to support kernel level memory page access control. Since PKS is

Listing 2: An example of context-dependent memory allocation

```
char *xreadline(int fd) {
    char *buf = malloc(MAX_LEN);
    for (int i = 0; i < MAX_LEN; i++) {
        int n = read(fd, buf+i, 1);
        if (n <= 0 || buf[i] == '\n') return buf;
    }
    return buf;
}
```
Fig. 2: MPK privilege switching at binary code level

the kernel-version of MPK, CRYPTO-MPK could be extended to protect crypto secrets in kernel easily. Similarly, CRYPTO-MPK could leverage the shred isolation primitives [23] to protect ARM executables.

3) Code Replication: CRYPTO-MPK generates different versions of code for different contexts. Algorithm 1 illustrates how CRYPTO-MPK creates multiple versions of code (functions) for different contexts. For instance, to protect the xreadline function in Listing 2, CRYPTO-MPK creates a protected version xreadline_r1 and replaces the invoking at Line 12, and a non-protected version xreadline_r2 for the invoking in the prompt function. Since Line 4 and 5 have been labeled as crypto operations, in xreadline_r1, CRYPTO-MPK inserts privilege switches before and after them. Moreover, by replacing malloc in Line 2 with a customized memory allocation function, CRYPTO-MPK reallocates buf to the protected memory region in xreadline_r1.

One problem of naive code replication is that it may lead to a significant code size bloating. As shown in Figure 3a, xreadline in context of main and readkey is same in transformation since their semantics in two contexts is identical. Thus generating two versions of these contexts are unnecessary. CRYPTO-MPK further introduces a function de-duplication algorithm to avoid generating redundant code (Algorithm 1). It works by calculating a signature for the transformation scheme of each context. Contexts of the same function with the same signature are considered identical and will share a single piece of code. In this way, code size bloating introduced by function replication is minimized.

The signature of a context is calculated as follows:

1) Collecting the information of the instructions operating on crypto buffers. Actual transformations will be done around these instructions (as defined in §III-D). For each of them, CRYPTO-MPK records its unique identifier (e.g. address) and its type (e.g. Load).

2) Collecting the signatures of sub-contexts. Two contexts can share the same code only if their callees are also the same. Thus for each CallInst, CRYPTO-MPK records its function call target is.

Algorithm 1: Function Replication

```
Data: Context set C in post-traversal order
Result: Output function set F
begin
  F ← {};
  forall c ∈ C do
    if c is root context then
      c.f ← c.f
    else
      S_f ← getFunctionVariants(c.f)
      s_c ← signature(c)
      if s_c ∉ S_f then
        c.f′ ← copyFunction(c.f)
        S_f ← S_f ∪ {s_c}
      else
        c.f′ ← variant in S_f with signature s_c
    end
  end
  forall f′ ∈ c.f do
    if f′ is a call by function pointer then
      replace f′ with direct function calls
    end
  end
  F ← F ∪ {c.f′}
end
```

3) Generating the final signature. Information from previous steps fully describes the transformation scheme of the context, thus CRYPTO-MPK uses hash_combine() in boost[24] to calculate its hash as the final signature.

IV. IMPLEMENTATION

A. Static Program Analysis

Static analysis is suitable for checking confidential data leaks [25]. CRYPTO-MPK relies on a context-sensitive points-to-analysis to identify crypto buffers and operations. Since CRYPTO-MPK aims to protect crypto operations comprehensively, it seeks to both accurately identify all secrets while only allowing relevant code to access them. Nevertheless, static points-to analysis is undecidable for C programs [26] and thus fundamentally imprecise [27], [28]. Consequently, CRYPTO-MPK modified the design of Dr.Checker [29] to implement a sounder analysis. The following design changes were made for the goal: 1) CRYPTO-MPK does not rely on depth-based heuristics or strong-update features to speed up the analysis, but keeps all points-to information; 2) CRYPTO-MPK adopts a more sophisticated design of alias objects to support points-to tracking in nested structures, global variables, and casted pointers, which are common in crypto applications but are oversimplified in Dr.Checker; 3) for dangling pointers with
which no object is found associated, CYMPK would fill in hollow objects to track their propagation thereafter.

To further ensure efficiency when analyzing complex crypto code bases, CYMPK employs two major strategies to counter the common challenges in a context-aware analysis. To address context explosion, it prunes off certain contexts where the corresponding functions are marked as orthogonal to the sensitive dataflow. To address points-to explosion, we made our observation that most of the objects are associated with memory pools (e.g. BN_POOL in OpenSSL [9]), and that they cannot be distinguished without proper abstraction [30]. Thus, CYMPK represents the pool with a single object without compromising the precision. On its finish, the static program analysis produces a tree of context objects, representing the call tree of the analyzed program. Each context object contains information on the taints and points-to relations among local values and allocated memory objects.

B. Labeling Crypto Buffers

We envision two types of users of CYMPK: (1) crypto library developers (the primary type) who are familiar with the logic of the various crypto functions and any sensitive data generated within the library, e.g., a private key. (2) application developers who use crypto libraries and are at least familiar with the APIs exposed by crypto libraries. This is important because sometimes crypto secrets can be generated by applications (host programs), e.g., supplying a password to a crypto hash function. Therefore, we currently also require application developers to label such secrets in order for CYMPK to track their propagation in the application context as well as the crypto library context. Later in §V-B and §V-C, we will report cases where applications-generated secrets do propagate in the application context (although not very often) and need to be protected. In both cases, CYMPK asks developers to annotate only the initial tags in the source code, which is shown to be small in both prior work [23] and our evaluation in §V-B. The propagation of tags is performed completely automatically by CYMPK during static analysis.

To annotate the initial tags, developers need to determine the crypto functions (either provided by the crypto library or directly implemented as a part of the host program), and identify the semantics of their input parameters to find key and plaintext/ciphertext. Next, developers could choose to directly annotate the key with a crypto tag and the plaintext/ciphertext with a mxor tag, or shift the annotation (and thus the protection) to earlier stage program inputs. For example, in Listing 1 developers could either label the rkk parameter of the xrijndaelEncrypt function with a crypto tag, or instead label the user input passwd string, which helps generate the AES key. In the future, we plan to automate this process by having crypto library developers annotate the outer most tags (e.g., parameters of a crypto API) and backtrack to the origin of such variables (possibly defined in the application context). That way, we can shift the burden of annotation to only the crypto library developers.

Listing 3: An example of initial tag annotating

```c
void do_encrypt() {
  #pragma tainter cryptotag(passwd)
  __cryptotag_fdf0f8a65855a52b(passwd);
  #pragma tainter mxortag(plaintext)
  __mxortag_fdf0f8a65855a52b(plaintext);
  encrypt(ciphertext, plaintext, passwd);
}
```
and many crypto libraries have already integrated cipher implementations specifically optimized for those instruction set extensions. We observed that at LLVM IR level, such an instruction (e.g., an AES-NI instruction) is implemented as an LLVM intrinsic function invoking. Therefore, CRYPTO MPK also handles such instructions as external functions with pre-defined rules.

C. Identifying Crypto Operations

After the crypto buffer labeling, CRYPTO MPK conducts a second code traversal to identify crypto operations. For each memory-related operation, if any of its operands is associated with a crypto tag, it is considered as a crypto operation.

External functions with pointer arguments, due to the lack of source code to analyze or modify, are also considered as memory operations. Ideally, we could conduct binary code analysis to analyze an external function/library and utilize a binary code rewriting to generate a protected version. We leave this as a future work of this research, and currently adopted a runtime permission granting mechanism for external functions (see Section IV-G).

D. Privilege Switch Instrumenting

Ideally, CRYPTO MPK inserts privilege switches right before and after a series of continuous crypto operation instructions to minimize the attack window. However, if we insert too many privilege switches, especially in hotspots, the execution will often suffer from performance issue (our micro-benchmark shows that the WRPKRU instruction takes about 20 to 30 CPU cycles, a lot more than an ordinary arithmetic instruction). To address, CRYPTO MPK trades off performance by protecting different instrumentation granularity against the code. For performance critical applications, CRYPTO MPK applies a function level crypto domain isolation for those hotspot functions. CRYPTO MPK considers the entire hotspot function as sensitive, and inserts privilege switches before and after any instructions that invoke it.

Hotspot functions are identified using a heuristic method. In general, CRYPTO MPK calculates a \( Q \) score for each context (Eq. 4), estimating the ratio of crypto operations within all memory operations. To resemble the case in real execution, instructions in loops are additionally weighted (Eq. 2), and numbers of instructions in sub-contexts are also estimated (Eq. 3). If the final \( Q_{ctx} \) is higher than a pre-defined threshold (0.25 here), CRYPTO MPK considers the function represented by \( ctx \) as a hotspot function. For recursive functions, due to their limited usage in crypto programs and the great analysis difficulty, CRYPTO MPK conservatively protects the entire recursive invocation.

\[
S_{inst} = \begin{cases} 
L_{inst} & \text{inst is a crypto operation} \\
L_{inst} * 30 & \text{inst is a callinst} \\
0 & \text{inst is a memory operation} \\n\end{cases} 
\]

\[
Q_{ctx} = \frac{\sum S_{inst} * T_{inst}}{\sum S_{inst}} 
\]

E. Invocation with Function Pointer

After CRYPTO MPK replicates a function, the next step is to replace all related invocations to guarantee the correctness of control flow. However, for a function invocation using function pointer, the control flow is often determined dynamically and we cannot directly rewrite the invocation. CRYPTO MPK therefore refers to a function pointer expanding technique used by Glamdring [31]. In our implementation, CRYPTO MPK first determines all possible candidates of a function pointer based on the results of points-to analysis, then replaces the function pointer invocation with expanded direct calls. In the example of Listing 4, CRYPTO MPK first evaluates the original invocation with \( funcptr \), finding two possible invoking targets (\( func1 \) and \( func2 \)). Then CRYPTO MPK adds two conditional invocations to the two replicated versions (\( func_r1 \) and \( func_r2 \)). For unexpected targets, they are invoked in a fallback branch without replication.

```c
- funcptr;/*args*/;
+ if (funcptr == func1)
  + func1_r1;/*args*/;
+ else if (funcptr == func2)
  + func2_r2;/*args*/;
+ else
  + funcptr;/*args*/;
```

Listing 4: Function pointer expanding

F. Crypto Buffer Management

To manage sensitive data in a unified way, CRYPTO MPK allocates all crypto buffers on a secure heap. It customizes the Jemalloc [32] allocator to support secure memory management. In detail, the customized allocator (namely \( m\_malloc \)) binds the allocated memory pages to a specific PKRU. Hence the access control of crypto buffers can be fulfilled efficiently by updating the PKRU register. In addition, it maintains a global table to record the pointer addresses of all crypto buffers, and replaces all memory de-allocation operations (including vanilla \( m\_free \)) with the unified \( m\_free \) function. When freeing a buffer using \( m\_free \), it checks the global table to choose the underlying de-allocator to use, either the customized version corresponding to \( m\_malloc \) for crypto buffers, or the standard \( m\_free \) for non-protected buffers.

Allocating all sensitive crypto buffers on our secure heap may introduce runtime overhead especially for stack memory operations. An optimization strategy adopted by CRYPTO MPK is that, if the length of a buffer is no more than a primitive scalar type (i.e. 8 bytes), CRYPTO MPK will not modify its stack memory allocation. Instead, CRYPTO MPK inserts memory zeroing code in function epilogue to clean
those small buffers immediately after the function execution. We argue that this kind of buffers seldom store long-term secret and a memory sanitization rather than secure isolation is able to defend against memory leak.

G. Runtime Checks

The static code analysis of CRYPTO-MPK may not perfectly identify all crypto operations due to the imprecision of points-to analysis. If a privileged operation is not correctly identified, a runtime exception will be triggered and the protected program will crash. To further prove the robustness, CRYPTO-MPK does not solely rely on static analysis but leverages a dynamic analysis to help find those operations. We argue that developers often possess enough test cases to execute the program. With these test cases, CRYPTO-MPK conducts an extra runtime test to check the protected program and finds potential exceptions.

In detail, CRYPTO-MPK links the protected program with a shared library that captures all runtime segment faults caused by MPK privilege violation. When the linked shared library captures a PKEY related segment fault during the test, it records the address of the fault instruction and the accessed memory buffer, and then temporarily enables access privilege for this instruction to continue the execution. After the entire execution, all unauthorized memory accesses are sent to developers for a manual verification. If the manual analysis confirms a memory access is actually legal (which implies that the accessed memory buffer should be labeled as crypto buffer but is missed by our static analysis), CRYPTO-MPK will automatically add a crypto tag for the related buffer and runs the static analysis again to identify more crypto operations.

In addition, if an unauthorized memory access occurs in an external library function (without source code and thus CRYPTO-MPK cannot rewrite it), CRYPTO-MPK links a specific segment fault handler to handle the exception, which restricts the range of code to access sensitive crypto buffers but suffers from performance slowdown.

V. EVALUATION

A. Experiment Setup

We use two sets of widely used open source projects to evaluate CRYPTO-MPK. The first set contains four popular crypto libraries including the famous OpenSSL [9] library, the libcrypt of GNU libc [33], and two modern crypto libraries, libodium [34] and libhydrogen [35], which aim to provide easy-to-use, hard-to-misuse functionalities. The second set contains four widely used Linux web server programs (Nginx [36] and Apache [10] as HTTP servers, vsftpd [37] as FTP server, and OpenSMTPD [38] as SMTP server), and crypt [11], a cryptographer developed cross-platform data encryption utility. Details of the tested libraries and programs are shown in Table IV in Appendix.

We ran our experiments on a server running Ubuntu 18.04 x64 with two Intel Xeon Gold 5122 Processors (each has eight logical cores at 3.60 GHz) and 128GB RAM. The current implementation of CRYPTO-MPK runs on top of LLVM 10.0.1.

In our experiment, we first utilized CRYPTO-MPK to analyze all programs, then we execute the analyzed programs with runtime checks instrumented, finally we generated executables to evaluate the performance overhead. In the following, we report our experimental results.

B. Annotated Tags and Costs

We tested CRYPTO-MPK by considering four typical application scenarios and the tested cases are detailed in Appendix. For all test cases, we evaluate the number of manually annotated tags and the approximate amount of effort expended for each program.

For Apache/Nginx+OpenSSL, we only need to annotated seven initial crypto tags in OpenSSL including six variables of the pkey struct that stores private key, and the AES_KEY key buffer of AES128-GCM API. We additionally annotated two initial mxor tags to plaintext buffer of RSA-sign and ciphertext buffer of RSA-verify, and plaintext/ciphertext buffers for AES128-GCM encryption/decryption, respectively.

For the message protection cases, we annotated three initial crypto tags to those crypto key parameters of both the used sign/verify APIs (e.g., hydro_sign_{create, verify} of libhydrogen) and the used encrypt/decrypt APIs (e.g., crypto_secretstream_xchacha20poly1305_sign, crypto_secretbox_encrypt of libhydrogen). After that, we annotated four initial mxor tags to plaintext/ciphertext parameters of those APIs.

Instead of annotating parameters of crypto functions, in the case of crypt encryption, we annotated four initial crypto tags to the buffer allocated by the readkey function in host program. This helps CRYPTO-MPK protect crypto secrets in the entire program. And we annotated three mxor tags to plaintext buffers, which are read from the input data file.

OpenSMTPD and vsftpd leverage the crypt password hashing function to transform user-provided passwords into a hash value. We observed the user-provided passwords are not only used by the crypt function but also other part of the host program, and thus we annotated the initial crypto tag to password buffers in host programs, and annotated the mxor tag to the salt parameters of the crypt API. An additional case is that for vsftpd with a parent thread and a child thread, we annotated two extra initial crypto tags separately to p_sess->ftp_arg_str of handle_pass_cmd in its parent thread, and password_str of process_login_req in the child thread. Therefore, CRYPTO-MPK could not only analyze the crypt function, but also track and protect the password propagation in these two host programs.

For each program, it took a single graduate student at most 10 to 15 hours (sometimes spread over multiple days) to understand where the labels should be assigned. We believe it

†To first generate analyzable non-optimized bitcode, we used the compilation option "-00 -xclang -disable-llvm-passes" to disable unnecessary optimizations. Next, the generated bitcode is sent to the analysis passes of CRYPTO-MPK to find sensitive buffers/operations and the output code is instrumented with our MPK-enabled protection. Finally, we could compile the bitcode into executables with LLVM backend (with the "-02" option to optimize the binary code).
TABLE I: Results of buffer labeling, operation identifying, and code protection

<table>
<thead>
<tr>
<th>Program</th>
<th>Tags</th>
<th>MemOps</th>
<th>Tainted MemOps (declassification)</th>
<th>Tainted MemOps</th>
<th>Total Functions</th>
<th>Crypto Functions</th>
<th># of Protected Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccrypt</td>
<td>4+3</td>
<td>1049</td>
<td>829</td>
<td>838</td>
<td>70</td>
<td>61</td>
<td>22(19)</td>
</tr>
<tr>
<td>libodium</td>
<td>1+1</td>
<td>6151</td>
<td>140</td>
<td>151</td>
<td>930</td>
<td>71</td>
<td>26(21)</td>
</tr>
<tr>
<td>libhydrogen</td>
<td>2+3</td>
<td>277</td>
<td>52</td>
<td>162</td>
<td>116</td>
<td>72</td>
<td>11(8)</td>
</tr>
<tr>
<td>Apache+OpenSSL</td>
<td>7+2</td>
<td>92279</td>
<td>2149</td>
<td>2528+2223</td>
<td>4879+11082</td>
<td>34+246</td>
<td>4+75(0+66)</td>
</tr>
<tr>
<td>Nginx+OpenSSL</td>
<td>7+2</td>
<td>70126</td>
<td>0+2149</td>
<td>4439+2223</td>
<td>1347+11082</td>
<td>24+246</td>
<td>0+75(0+66)</td>
</tr>
<tr>
<td>OpenSMTPD+ccrypt</td>
<td>2+1</td>
<td>14170</td>
<td>4+492</td>
<td>7+566</td>
<td>892+40</td>
<td>22+33</td>
<td>4+18(0+16)</td>
</tr>
<tr>
<td>vsftpd+ccrypt</td>
<td>3+1</td>
<td>9122</td>
<td>9+492</td>
<td>10+566</td>
<td>5964+90</td>
<td>42+33</td>
<td>9+18(0+16)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>39</td>
<td>194,036</td>
<td>6,321</td>
<td>13,713</td>
<td>31,074</td>
<td>396+40</td>
</tr>
</tbody>
</table>

HF: Hotspot functions; AP: accurately protected; OP: over protected; UP: under protected; RF: replicated functions for different contexts

will take developers of the application or crypto libraries even less time to perform the labeling.

C. Labeling Accuracy

To evaluate whether CRYPTO MPK identifies intermediate crypto data and relevant operations accurately, we first executed the protected programs (instrumented with runtime checks) using previously prepared test cases. We found that the executed programs passed all test cases without producing a feedback, which indicates that CRYPTO MPK properly granted permissions to all memory accesses of crypto buffers in tested programs. Then we manually verified the overall labeling results, which are listed in Table I. We observed that both the identified crypto memory operations (Tainted MemOps (declassification)) and functions to fulfill crypto operations (crypto functions)) are only a small portion of those in the entire programs. Agadakos et al. [39] also reported a similar result, which shows 93.82% of code in OpenSSL-libcrypto could be removed in specific application scenarios. This demonstrated the in-process isolation among used and non-used code is expected to reduce attack surfaces significantly.

We found for ccrypt, libodium, libhydrogen, and crypt with OpenSMTPD and vsftpd, the ciphertext does not get further propagated, then the effect of over-tainting is moderate (Tainted MemOps vs. Tainted MemOps (declassification)). However, if the ciphertext is further used by other parts of the application (which is the case for Apache and Nginx), we saw a substantial increase of sensitive memory operations that we have to protect, i.e., 2528/4439 extra memory operations. Actually, Apache and Nginx do not access the crypto secret. They delegate the management of crypto buffers to crypto libraries instead. In these cases, the crypto-aware analysis successfully excludes unnecessary code in host programs from accessing sensitive crypto secrets. CRYPTO MPK also labeled four and nine sensitive memory operations in the application contexts for OpenSMTPD and vsftpd, respectively. We further confirmed these operations in the host programs access the passed buffers and thus should also be protected.

Similar to CRYPTO MPK, DynPTA [40] labeled 12.79% memory operations as sensitive for Nginx+OpenSSL, while CRYPTO MPK labeled only 3.06% (2149/70126). Our extended analysis further demonstrated that had CRYPTO MPK not adopted the crypto-aware analysis (declassification), the labeled memory operations would increase to (4439+2223)/70126=9.5%, which is consistent with the result of DynPTA.

Among all crypto functions, CRYPTO MPK found 296 (34.3%) of them accessed crypto secrets. As Table I shows, except four functions in ccrypt and libodium, CRYPTO MPK accurately protected functions that operated crypto buffers. We additionally examine the over protected (OP) and under protected (UP) functions. We found those functions involve either an external file reading or a random number generation. For file reading functions, CRYPTO MPK could not determine whether the read content is sensitive and thus under protect them. For random number generation functions, CRYPTO MPK considers all random numbers used in a crypto application are sensitive and thus over protect them.

CRYPTO MPK conducted a function level crypto domain isolation for 212 of the 296 protected functions, and conducted instruction level isolation for 84 functions. It followed the strategy mentioned in Section IV-D to choose the appropriate protection granularity. We further invited two experienced players who have participated in DEF CON CTF Finals to help check these functions, and they did not find vulnerable code in those functions. We also counted the number of replicated functions in the last column (RF) of Table I to show the necessity of our context-sensitive analysis. The number indicates for all protected functions (Total(HF)), how many different contexts are involved (and thus we need to generate a new copy of the function). This allows a fine-grained protection so that we don’t have to choose between either “always protect” or “always not protect” a function.

D. Security Analysis

To model the capabilities of attackers who can execute memory corruption based information leak attacks, we adopt the taxonomy proposed by Szekeres et al. [41]. By utilizing either an out-of-bounds pointer or a dangling pointer, attackers could access unauthorized memory regions. In response, we illustrate how CRYPTO MPK defends against attacks using such primitives.

- **Out-of-bounds Pointers.** We assume that an attacker owns an out-of-bounds pointer in non-crypto domain. He can use this pointer to either read data before or after a specific memory region (buffer overflow or underflow respectively), and the pointer can even be used to read content of a designated memory address (arbitrary memory read).


For CRYPTO MPK protected programs, the isolation guarantees that the operations in crypto domain do not use the out-of-bounds pointer in non-crypto domain. Even if the attacker controls this pointer and could assign it with any memory address of crypto buffers, he cannot use this pointer to access the protected buffers because he lacks the corresponding PKEY permission. Therefore, the attacker could not exploit such vulnerable pointers to leak sensitive information.

- **Dangling Pointers.** Similarly, we assume that an attacker owns a dangling pointer in non-crypto domain. CRYPTO MPK guarantees that this pointer should never point to a crypto buffer because CRYPTO MPK uses two independent memory managers for crypto domain and non-crypto domain, respectively. A dangling pointer only points to memory buffers of non-crypto domain and thus the attacker could not leak sensitive data using this vulnerable pointer.

Besides confidentiality protection, we also enhance the integrity protection by adding the binary code rewriting of ERIM [42] to eliminate any unintentional occurred WRPRRU or XRSTOR gadgets, and thus prevent illegal privilege switching caused by code reuse attacks.

### E. Protection Effectiveness

We checked the results of the analysis-driven labeling of CRYPTO MPK, and found it protected not only crypto keys but also confidential information propagated from the secret keys. We manually examined the protected buffers of each crypto applications to verify whether crucial propagated secrets were protected (details of CRYPTO MPK labeled code can be downloaded at https://cryptompk.code-analysis.org). For all tested applications, we found propagated secrets were well-protected:

- **SSL/TLS.** Besides private key and session key, CRYPTO MPK additionally protected crypto operations in 75 functions of OpenSSL, e.g., the RSA sign function uses Montgomery modular multiplication [43] to accelerate exponential operation. In this function, the Montgomery transformation of private key components \( p \) and \( q \) produces several intermediate \( \text{BIGNUM} \) structs, and those buffers are kept on the heap for a long time.

- **AES.** CRYPTO MPK labeled crypto buffers on either stack or heap for ccrypto. In particular, ccrypto first reads a variable-length key passphrase from the external input, copies the initial secret to a password buffer with specific newline format key2, then hashes it into a 16-byte keyblock. Next, it executes the standard AES key scheduling to extend the 16-byte keyblock to 15 round keys stored in an AES round key array \( \text{rkk} \) (each round key is a 16-byte array). Moreover, all these buffers are not sanitized after the crypto operations. As a result, a memory disclosure of either key2, keyblock, or \( \text{rkk} \) buffer could give the attacker enough information to recover the AES key and thus decrypt the ciphertext.

- **Chacha20-Poly1305.** Among the labeled crypto buffers of libsodium and libhydrogen, we found specific crypto context structs (e.g., poly1305_state for the chacha20-poly1305 cipher). In further, some functions would propagate the secret key information from a context struct to a local variable. For instance, in the `crypto_secretstream_xchacha20poly1305_rekey` function of libsodium, the secret information propagates to the `new_key_and_inonce` array on the stack. If this array is leaked, the attacker only needs to guess a nonce offset (corresponding to the size of ciphertext) by brute-force to decrypt the encrypted data.

- **Crypt-SHA512.** With the labeled tags, CRYPTO MPK traversed the code and discovered crypto data in 22 and 27 functions in OpenSMTPD and vsftpd, respectively. An interesting observation is that the developer of crypt tried to clean sensitive buffers after the hashing (e.g., the `free_key` buffer). However, this manual sanitization is far from comprehensive and there still exist many unprotected buffers. For instance, the 64-byte `correct_words` array in the API is an intermediate buffer that stores input password and will not be sanitized after the execution. Once the attacker obtains this information, he can then recover the input password.

To prove the necessity of our protection, we additionally chose the following five memory disclosure CVEs to test our protection against different types of memory issues:

- **CVE-2011-4576:** an uninitialized variable vulnerability in OpenSSL which leads to a not properly initialized block cipher padding, and attackers could obtain at most 8191 bytes of uninitialized data on the heap.

- **CVE-2014-0160:** the well-known *HeartBleed* bug of OpenSSL that allows remote attackers to obtain sensitive information from process memory via crafted packets.

- **CVE-2016-2176:** a vulnerability that affects `XS09_NAME_oneline` function in OpenSSL. Attackers could use this vulnerability to fulfil out-of-bounds reads and obtain at most 1024KB data on the stack.

- **CVE-2017-9798:** a use-after-free vulnerability in Apache. When attackers send an OPTIONS HTTP request, the server would read data from a freed chunk and may lead to potentially sensitive information disclosure.

- **CVE-2018-16845:** an integer overflow vulnerability in Nginx. Attackers can use a compromised variable to execute out-of-bounds memory access.

Note that the vulnerable versions of Apache, Nginx, and OpenSSL are not compatible with each other, we simulated attacks by first porting those CVEs to our protected versions of Apache/Nginx+OpenSSL (but ONLY protected the annotated secrets), and then utilizing the vulnerabilities to search propagated secrets in process memory. Even though an attack could not directly read the crypto key, it could still retrieve a plentiful of propagated secrets from the process memory and leverage those information to recover the key. For instance, an attack could utilize several of the above mentioned CVEs to access heap buffers and obtain intermediate values of Montgomery modular multiplication. By leaking a special intermediate big number \( RR (RR \equiv 2^p \mod p, p \text{ is one of the secret large primes used in RSA cryptosystem}) \), the attacker could calculate \( p \) and then factor the RSA big number to obtain another prime \( q \), and finally recover the entire private key triple \((d, p, q)\) by calculating the modular inverse \( d \).
of $e \mod (p - 1)(q - 1)$. Similarly, attackers could recover AES and Chacha20 keys using intermediate states, and use the intermediate blocks of crypt-SHA512 to recover input password. In response, CRYPTO MPK extensively identifies intermediate crypto buffers and relocates them to protected region, and guarantees the security since memory disclosure attacks cannot recover useful secrets in non-crypto domain.

F. Performance Tests

1) Runtime Overhead: We tested the runtime overhead introduced by CRYPTO MPK by evaluating the performance of protected OpenSSL, libsodium, and libhydrogen libraries. In detail, we tested the slowdown for RSA and AES using OpenSSL speed test (a built-in cipher benchmark of OpenSSL), and the overhead of real-world crypto applications by 1) testing HTTPS throughput of Nginx/Apache+OpenSSL with the ApacheBench [44] (both programs were configured to use ECDHE-RSA-AES128-GCM-SHA256 cipher suite, ApacheBench simulated 20 clients, each sending 1,000 requests); 2) encrypting 100 MB random data with libsodium; 3) signing/encrypting 100 MB random data using libhydrogen.

Figure 4 depicts the overhead for each test case. In Figure 4a and Figure 4b, we found except the less frequently used key-size (RSA-1024) or input-size (AES-GCM encryption with a 16-byte input), the execution slow down is moderate (from 7.13% to 12.79%). This demonstrates the performance impact for most computational intensive crypto algorithms is acceptable. In comparison, Glamdring [31] with a similar code partitioning model suffered from more than 30% slowdown for RSA-4096 sign and about 40% slowdown for AES-CBC-256.

For real-world crypto applications (Figure 4c), we observed a low overhead for a fine-grained isolation: The data encryption with libsodium and libhydrogen suffered 9.53% and 4.52% extra execution time, respectively. And the public key signing of data using libhydrogen only incurred an overhead of 1.07%. In particular, the overhead of HTTPS communication under Apache Benchmark is only 6.25% and 5.19% for Nginx and Apache, respectively. Compared to CRYPTO MPK, ERIM [42] and libmpk [5] that also utilize MPK to protect Nginx+OpenSSL (only protecting the AES session keys) and Apache+OpenSSL (only protecting the RSA private key) introduced an overhead of 5.06% and 4.82%, respectively. Since CRYPTO MPK additionally protects the RSA private key as well as 2,149 memory operations in this case, its MPK-enabled crypto domain isolation is very efficient.

2) Code Bloating: We tested the code bloat effect brought by the code transformation of CRYPTO MPK. Particularly, we measured how crypto code (libraries such as OpenSSL and crypto utilities such as ccrypt) bloat after the protection. After applying CRYPTO MPK to targets listed in Figure 5, we found the code bloat for crypto code is acceptable for cloud servers. For larger targets (OpenSSL and libsodium), the code bloating ratios are less than 20%. For targets with relatively small code base (libhydrogen, libcrypt, and ccrypt), although the code bloating ratios exceed 50%, the actually increased code size is less than 50 KB.

Since the code bloat is mainly due to the function replication (Section III-E3), we counted the number of copied functions (not only the MPK protected ones shown in Table I, but also their callers) for each tested target. We found although the bloating ratios of libhydrogen, libcrypt, and ccrypt are more the 50%, the number of copied functions (59 for libhydrogen, 29 for libcrypt, and 64 for ccrypt) is not large. For libraries with larger code base, the copied functions only occupied a small portion. For instance, CRYPTO MPK only copied 184 of 11,082 (4% of the entire code size) functions in OpenSSL.

3) Analysis Costs: We evaluated the costs of static source code analysis by measuring the consumed time of crypto buffer labeling and crypto operations identification. The results are shown in Table II. Note that for Apache and Nginx, they executed the same APIs of OpenSSL to enable TLS and for each host program only one function is involved. Thus we only consider the analysis time against the code of OpenSSL.

From Table II we observed that CRYPTO MPK could accomplish the analysis of most targets in less than three seconds. The only exceptions are the analysis against the RSA implementation of OpenSSL and the password hashing of vsftpd: it took hundreds of seconds for CRYPTO MPK to finish the crypto buffer labeling/crypt operations identification tasks. We checked the analysis procedure and found the reasons.
TABLE II: Analysis performance for each experiment target

<table>
<thead>
<tr>
<th>Case</th>
<th>SBL(s)</th>
<th>SMOI(s)</th>
<th>Total(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>libhydrogen</td>
<td>0.31</td>
<td>0.41</td>
<td>0.72</td>
</tr>
<tr>
<td>ccrypt</td>
<td>0.97</td>
<td>1.02</td>
<td>1.99</td>
</tr>
<tr>
<td>OpenSMTPD+libcrypt</td>
<td>1.41</td>
<td>1.02</td>
<td>2.43</td>
</tr>
<tr>
<td>libodium</td>
<td>2.04</td>
<td>1.81</td>
<td>3.85</td>
</tr>
<tr>
<td>vsftpd+libcrypt</td>
<td>244.39</td>
<td>52.91</td>
<td>297.30</td>
</tr>
<tr>
<td>(Apache, Nginx)+OpenSSL</td>
<td>638.29</td>
<td>540.94</td>
<td>1179.23</td>
</tr>
</tbody>
</table>

SBL : time of crypto buffer labeling;
SMOI : time of sensitive memory operation identification

For case of OpenSSL, the involved code in OpenSSL is significantly more complex than that of other targets. A particular reason is that OpenSSL implemented more than one versions of modulus and exponent algorithms for different platforms. CRYPTO MPK, however, is path-insensitive and traversed all of them and thus spent a longer analysis time. For vsftpd, the major overhead is due to the analysis of the host program (i.e., vsftpd) rather than the crypt API. Among the 596 functions of vsftpd, 9 of them involve crypto operations, and vsftpd uses two threads to manage crypto secrets separately. Therefore CRYPTO MPK spent more analysis time. In comparison, the similar analysis against crypt with OpenSMTPD only spends 2.43 seconds in total, since the host OpenSMTPD program adopts a much simpler way to operate crypto secrets.

VI. RELATED WORK

Keeping crypto keys unprotected in memory is dangerous. Attackers could recover the key through code injection or side channel attacks [19], [45]. However, not all memory buffers holding crypto keys are properly sanitized [46]. Actually, K-Hunt [47] examined popular crypto libraries and utilities, and found that recoverable keys widely existed. Micheli et al. demonstrated crypto keys could be effectively recovered from partial information [48]. CRYPTO MPK advances state-of-the-art by considering not only the crypto keys but also their derived secrets. To the best of our knowledge, we are the first to systematically label all kinds of crypto secrets in programs.

A large number of isolation solutions have been proposed to isolate confidential data in memory. We summarized nine most relevant systems and compare them with CRYPTO MPK in Table III. Among them, ContLLVM, libmpk, xMP, Seimi, and Shreds only provide isolation primitives, and require developers to manually annotate sensitive data and operations. With only a manual annotation it is difficult to cover all propagated secrets. To automate sensitive data labeling, Cali, Datashield, Glamdring, and SeCage utilize different kinds of data-flow analyses. In particular, Cali leverages program dependency graph to infer the sensitive memory buffers; Datashield utilizes inter-procedural and context-sensitive dataflow analysis to find all the explicitly and implicitly sensitive variables; Glamdring utilizes static data-flow analysis and static backward slicing to find control and data dependencies on the sensitive data annotated by developers; SeCage combines static and dynamic analysis to decompose monolithic software into several compartments and isolates sensitive data. Unfortunately, none of them considers the characteristic of crypto operations to avoid over-tainting issues. In comparison, CRYPTO MPK implements a fine-grained labeling that are both context-sensitive and crypto-aware to guarantee the precise isolation of crypto secrets.

From the perspective of isolation mode, We categorize systems in Table III in address-based isolation and domain-based isolation groups. Corresponding to their isolation mode, these systems leverage different software and hardware-featured primitives to implement a more efficient isolation. ContLLVM and Datashield adopt the Intel MPX feature to help isolate sensitive data based on an address-based isolation. For SeCage, SeIMI, and xMP that utilizes different hardware features, they all need to modify system kernel to implement the protection. For Cali and Glamdring, they require to split the process into multiple parts. In comparison, only libmpk and CRYPTO MPK seldom modifies the execution model and are easy to deploy.

There exist some orthogonal approaches that would enhance the protection of CRYPTO MPK. We have in fact applied ERIM [42] to enhance the protection of CRYPTO MPK. MOSA [49] uses hardware transactional memory (HTM) to ensure that crypto keys are never loaded to RAM chips. It however only protects private keys, and does not track and protect the derived secrets especially those long-term buffers, which is a key difference from CRYPTO MPK. Wedge [50] and ProgramGutter [51] partition software into least privilege components, and PM [52] could automatically find partitions that have better balance between security and performance. These techniques could retrofit the domain model of CRYPTO MPK (e.g., supporting more than one crypto domain).

VII. CONCLUSION

Protecting confidential data against memory disclosure attacks is crucial to crypto applications. In this paper we present CRYPTO MPK, which automates crypto secrets tracking and implements fine-grained protection for server applications. With the help of CRYPTO MPK, even a non-expert developer could easily build and deploy binary executables with crypto secrets isolated. Our evaluation show that CRYPTO MPK provides reliable protection against memory disclosure attacks, and the protection only incurs at most 9.53% runtime overhead for widely used crypto applications.

TABLE III: Comparison related systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cali [53]</td>
<td>✓</td>
<td>✗</td>
<td>OP</td>
<td>DI</td>
<td>Isjail [54]</td>
</tr>
<tr>
<td>ContLLVM [8]</td>
<td>✗</td>
<td>✓</td>
<td>UP</td>
<td>AI</td>
<td>MPX [55]</td>
</tr>
<tr>
<td>Datashield [6]</td>
<td>✓</td>
<td>✓</td>
<td>OP</td>
<td>AI</td>
<td>SGX [56]</td>
</tr>
<tr>
<td>Glamdring [31]</td>
<td>✓</td>
<td>✓</td>
<td>OP</td>
<td>DI</td>
<td>SGX [56]</td>
</tr>
<tr>
<td>SeCage [7]</td>
<td>✓</td>
<td>✓</td>
<td>OP</td>
<td>DI</td>
<td>VMMFUNC [57]</td>
</tr>
<tr>
<td>SEIMI [55]</td>
<td>✓</td>
<td>✓</td>
<td>UP</td>
<td>DI</td>
<td>SMAP [59]</td>
</tr>
<tr>
<td>Shreds [23]</td>
<td>✗</td>
<td>✗</td>
<td>UP</td>
<td>DI</td>
<td>ARM Memory Domains [60]</td>
</tr>
<tr>
<td>xMP [4]</td>
<td>✗</td>
<td>✗</td>
<td>UP</td>
<td>DI</td>
<td>EPTP [61]</td>
</tr>
</tbody>
</table>

CRYPTO MPK ✓ ✓ AP DI MPK

C1: Automated Annotation
C2: Context-sensitive Protection
C3: Propagated Secrets Protection
C4: Isolation Mode
C5: Isolation Feature
AI: Address-based Isolation
DI: Domain-based Isolation
UP: Under-protected
AP: Accurately-protected
OP: Over-protected
ACKNOWLEDGEMENT

The authors would like to thank the reviewers for their valuable feedback during the revision process. This work was partially supported by the National Natural Science Foundation of China (No.62002222), the National Key Research and Development Program of China (No.2020AAA0107800), and the start-up funding of the School of Information Technology and Electronic Engineering, the University of Queensland. We especially thank Ant Group for the support of this research within the SJTU-Ant Security Research Centre.

REFERENCES

A. Details of Test Cases

We used two sets of widely used open source projects to evaluate CRYPTO MPK. The first set contains four crypto libraries and the second set contains four widely used Linux web server programs. Details of the tested libraries and programs are shown in Table IV. The source code of them could be found in https://cryptompk.code-analysis.org. To evaluate whether CRYPTO MPK can be applied to protect sensitive data in real world applications, we consider four typical application scenarios and tested CRYPTO MPK with the corresponding cases.

1) File Encryption. We employed crypt to conduct file encryption/decryption. crypt reads the passphrase from a local file and derives its AES encryption key. Then it uses the derived key to fulfill file encryption/decryption.

2) Message Protection. We implemented an authentication encryption program using ciphers of libsodium and libhydrogen (i.e., xchacha20-poly1305 stream cipher and Curve25519 elliptic curve public key cipher), which fulfills message encrypting and signing.

3) SSL/TLS Transmission. We deployed Nginx and Apache with OpenSSL to provide HTTPS service for browsers. We chose the widely used ECDHE-RSA-AES128-GCM-SHA256 cipher suite for the TLS connection. Note that the analysis for other crypto suites are similar and thus we only tested one of them.

4) Password Authentication. We deployed vsftpd and OpenSMTPD with password authentication to provide FTP and SMTP services, respectively. In both applications, the input passwords are verified by the crypt password hashing API provided by libcrypt of glibc.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>Line of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>libcrypt</td>
<td>2.27</td>
<td>3,016</td>
</tr>
<tr>
<td>libhydrogen</td>
<td>0.2.0</td>
<td>2,597</td>
</tr>
<tr>
<td>libsodium</td>
<td>1.0.18</td>
<td>32,279</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>1.0.2u</td>
<td>163,734</td>
</tr>
</tbody>
</table>

TABLE IV: The evaluated programs and crypto libraries.

B. Compiling and Analyzing Experimental Targets

In the following, we briefly present the steps of how to compile and analyze our experimental targets. More detailed instructions could also be found in https://cryptompk.code-analysis.org.

To analyze the source code of the tested projects, CRYPTO MPK needs to first compile source code into LLVM IR bitcode for the following analyses. To implement a cross-module analysis, CRYPTO MPK merges LLVM IR bitcode modules generated from each source code file into a single bitcode file. It first utilizes the LLVM compiler infrastructure to handle source code in the pre-processing phase (by using the -emit-llvm option of Clang [62]) to generate a set of LLVM IR bitcode modules. Next, CRYPTO MPK merges them into a single bitcode file by leveraging LLVM gold plugin [63].

CRYPTO MPK by default adopts a conservative optimization level (i.e., -00 for LLVM) to generate bitcode. Using such a compilation option, all data references in bitcode are in the form of memory reference. We observe that a large number of memory references often lead to unacceptable memory overhead for CRYPTO MPK in sensitive buffer labeling phase. As a result, CRYPTO MPK further applies the mem2reg optimization pass to transform memory references of function primitive type data to register references. CRYPTO MPK also applies jump threading, simplifycfg and dce passes to optimize control flow, which will improve analysis performance.

When applying CRYPTO MPK to complex code bases such as OpenSSL and vsftpd, some execution environments with limited RAM may suffer from out-of-memory issue. We
TABLE V: Performance overhead comparison between CRYPTO/MPK fully-protected programs and their memory re-allocation only counterparts

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Baseline</th>
<th>Fully Protected</th>
<th>Memory Re-allocation Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA1024-Sign (sign/s)</td>
<td>2.264.19</td>
<td>1.933.86 (14.59%)</td>
<td>2.066.51 (8.73%)</td>
</tr>
<tr>
<td>RSA2048-Sign (sign/s)</td>
<td>373.92</td>
<td>337.52 (9.73%)</td>
<td>344.24 (7.94%)</td>
</tr>
<tr>
<td>RSA5066-Sign (sign/s)</td>
<td>55.29</td>
<td>51.06 (7.66%)</td>
<td>51.44 (6.96%)</td>
</tr>
<tr>
<td>RSA1024-Verify (verify/s)</td>
<td>50.548.27</td>
<td>43,629.8 (13.69%)</td>
<td>44,251.45 (12.46%)</td>
</tr>
<tr>
<td>RSA2048-Verify (verify/s)</td>
<td>14,399.73</td>
<td>12,557.67 (12.79%)</td>
<td>12,612.88 (12.41%)</td>
</tr>
<tr>
<td>RSA4096-Verify (verify/s)</td>
<td>3,715.38</td>
<td>3,295.75 (11.29%)</td>
<td>3,298.49 (11.22%)</td>
</tr>
<tr>
<td>AES-16 (KB/s)</td>
<td>113,876.33</td>
<td>90,228.85 (20.77%)</td>
<td>103,922.18 (8.74%)</td>
</tr>
<tr>
<td>AES-64 (KB/s)</td>
<td>119,801.85</td>
<td>107,516.41 (10.25%)</td>
<td>110,329.25 (7.91%)</td>
</tr>
<tr>
<td>AES-256 (KB/s)</td>
<td>122,768.49</td>
<td>113,073.89 (7.90%)</td>
<td>113,779.24 (7.32%)</td>
</tr>
<tr>
<td>AES-1024 (KB/s)</td>
<td>123,387.07</td>
<td>114,451.00 (7.24%)</td>
<td>114,776.03 (6.98%)</td>
</tr>
<tr>
<td>AES-8192 (KB/s)</td>
<td>123,664.66</td>
<td>114,844.60 (7.13%)</td>
<td>115,042.03 (6.97%)</td>
</tr>
<tr>
<td>Nginx (ms/req)</td>
<td>109.14</td>
<td>114.81 (5.19%)</td>
<td>114.03 (4.48%)</td>
</tr>
<tr>
<td>Apache (ms/req)</td>
<td>1.102</td>
<td>1.207 (9.53%)</td>
<td>1.204 (9.26%)</td>
</tr>
<tr>
<td>libhydrogen-enc (sec)</td>
<td>3.122</td>
<td>3.263 (4.52%)</td>
<td>3.189 (2.15%)</td>
</tr>
<tr>
<td>libhydrogen-sign (sec)</td>
<td>1.686</td>
<td>1.704 (1.07%)</td>
<td>1.696 (0.59%)</td>
</tr>
</tbody>
</table>

suggest that analysts could choose to ignore a certain group of functions to finish the analysis: (i) ASN1-family functions in OpenSSL are related to parsing ASN1 certificate files. Such parsing logic is very complicated, and can leads to path explosion in analysis if not handled. Since they do not propagate any taint tags based on our knowledge, we pruned them off in the analysis; (ii) the built-in dynamic memory management functions in OpenSSL. OpenSSL uses BN_P00L_get to implement stack-like allocation of BIGNUM. However, the static analysis of CRYPTO/MPK cannot deal with memory pool well for CRYPTO/MPK is adopting weak-updating strategy as mentioned in Section IV-B. Hence CRYPTO/MPK handles BN_P00L_get specifically. In sensitive buffer labeling phase, CRYPTO/MPK only allocates object at the first time it traverse into BN_P00L_get. (iii) Error handling function is frequently used in vsftpd, which introduce high analysis costs for CRYPTO/MPK. Empirically, those functions are not vulnerable, we manually disable the analysis against them (e.g. bug d1e and d1e2) to improve the analysis performance.

After CRYPTO/MPK finishes all LLVM IR level code transformation for crypto buffer protection, it performs a final pass to optimize the generated IR instructions. Then, the optimized IR code are compiled into binary code, and linked it to a pre-built shared library that provides secure memory management (i.e., implementing malloc and free based on Jemalloc) to generate the executables, and we could utilize the binary analysis module of ERIM to check the generated binary code.

C. Evaluation for Performance Overhead of Memory Re-allocation and CFI

Since the overhead of CRYPTO/MPK is not only caused by MPK privilege switching but also by our implemented memory re-allocation, we further measured the runtime performance of tested targets with memory re-allocation only (i.e., disabling MPK privilege switching), and compared it with the performance of fully-protected targets. The result is shown in Table V. We found that the introduced memory re-allocation actually contributed a major part of the entire overhead in most test cases, and this indicates that a further optimized memory re-allocation mechanism could be fulfilled to implement a more efficient protection. We leave this as a future work.

The overhead of CRYPTO/MPK is only related to memory isolation and does not involve integrity protection. To approximately evaluate the overhead of an integrated solution combining CRYPTO/MPK and CFI, we measured the combined performance overhead with both CRYPTO/MPK and the LLVM-CFI [64]. Note that the used LLVM-CFI (version 10.0.1) still introduced compatibility issues for OpenSSL since the crypto library contains many cases of mismatching between a function pointer and its pointed function. We therefore manually disable the CFI for those function pointers in our experiments.

As shown in Table VI, after applying CFI protection, the runtime overhead additionally increased around 2% for our tested targets. This implies the feasibility of combining current CFI mechanisms and CRYPTO/MPK. Nonetheless, we point out that ensuring integrity is still a very challenging problem and is under active research. As a result, CRYPTO/MPK still focuses on the confidentiality aspect only in this paper.

TABLE VI: Results of MPK+CFI Protection

<table>
<thead>
<tr>
<th>Program</th>
<th>MPK overhead</th>
<th>MPK+CFI overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>libhydrogen</td>
<td>11.37%</td>
<td>13.74%</td>
</tr>
<tr>
<td>libhydrogen-enc</td>
<td>4.52%</td>
<td>4.52%</td>
</tr>
<tr>
<td>libhydrogen-ign</td>
<td>1.07%</td>
<td>1.07%</td>
</tr>
<tr>
<td>Apache+OpenSSL</td>
<td>5.19%</td>
<td>7.03%</td>
</tr>
<tr>
<td>Nginx+OpenSSL</td>
<td>6.62%</td>
<td>7.69%</td>
</tr>
</tbody>
</table>