AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content

Shruti Tople Shweta Shinde Zhaofeng Chen Prateek Saxena
School of Computing
National University of Singapore
{shruti90, shweta24, chenzhao, prateeks} @comp.nus.edu.sg

ABSTRACT

Web servers are vulnerable to a large class of attacks which can allow a network attacker to steal sensitive web content. In this work, we investigate the feasibility of a web server architecture, wherein the vulnerable server VM runs on a trusted cloud. All sensitive web content is made available to the vulnerable server VM in encrypted form, thereby limiting the effectiveness of data-stealing attacks through server VM compromise.

In this context, the main challenge is to allow the legitimate functionality of the untrusted server VM to work. As a step towards this goal, we develop a tool called AUTOCRYPT, which transforms a subset of existing C functionality in the web stack to operate on encrypted sensitive content. We show that such a transformation is feasible for several standard Unix utilities available in a typical LAMP stack, with no developer effort. Key to achieving this expressiveness over encrypted data, is our scheme to combine and convert between partially-homomorphic encryption (PHE) schemes using a small TCB in the trusted cloud hypervisor. We show that x86 code transformed with AUTOCRYPT achieves performance that is significantly better than its alternatives (downloading to a trusted client, or using fully-homomorphic encryption).

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Processors—*Compilers*; D.4 [**Operating Systems**]: Security and Protection—*Cryptographic Controls*

Keywords

Web Security; Homomorphic Encryption; Type Systems

1. INTRODUCTION

Web hosting services, such as RackSpace and Amazon Web Services, enable web server stacks (e.g. a LAMP web server) to be hosted on public clouds using VMs. User and enterprises often wish to maintain strong data protection of sensitive data stored on public cloud servers from web and network attacks. Web server stacks are prone to a large class of attacks ranging from SQL injection, memory corruption vulnerabilities, OS command injection, server misconfiguration, file type confusion bugs, and so on. These attacks can be used to compromise web servers and install malware. Despite heightened security concerns [10], stronger data protection laws [4, 8, 9], and availability of commercial detection tools [6, 7], server-side data breaches have been persistently high for the last 3 years [25].

Previous work has proposed partitioning monolithic web servers into multiple pieces, as a second line of defense. For instance, separating the web application logic into multiple VMs based on roles [49], privilege separating users using OS protections [39], or using trusted hardware features to attest the integrity of server VMs [44, 45, 61]. In this work, we advocate a new approach for building a second line of defense: we investigate how to protect sensitive web content in a compromised web server (running in a VM) on a trusted cloud. Specifically, we focus on ensuring the confidentiality of sensitive file-based web content while still allowing legitimate applications to operate on them.

Problem Setting. In this paper, we investigate the feasibility of protecting sensitive web content transparently in an untrusted server VM on a trusted cloud provider. We assume that the cloud provider and underlying VMM / hypervisor is trusted, but the attacker can exploit a vulnerability in the server-side web stack to run arbitrary code on the web server VM. One promising approach to achieve this is to encrypt all sensitive file content before exposing it to the untrusted server VM [52]. With this mechanism, attacks through the VM are forced to access information in encrypted form and computationally bounded adversaries cannot learn the information in sensitive files. Sensitive content is encrypted outside the untrusted VM (e.g. on a separate trusted key server or on the client's device) and encryption keys are made available only to the trusted hypervisor. A compromised VM would only be able to leak encrypted content to remote adversaries (which don't have the decryption keys). In this conceptual setting, the hypervisor acts as a static root-of-trust and does not execute any unprivileged code from the untrusted VM in privileged land. We discuss the merits of this conceptual architecture in Section 2 in more depth to motivate the context and need for our auto-transformation tools.

Challenge & Approach. The main challenge in enabling this defense is that naïvely encrypting content in the untrusted VM disables all legitimate (or benign) functionality. The standard solution for this problem is to download the encrypted data on the client side and then compute on the decrypted data. We call this approach as the 'download-and-compute' mechanism. Indeed, several commercial solutions — such as BoxCryptor [18] and CloudFogger [21] — allow users to store encrypted data on the server and only permit operating on the data by downloading full contents to the trusted client device before computation. However, for large amounts of data (say over a 1 TB of logfile), downloading the data incurs huge network communication which has direct financial costs for the

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516666 .

service provider, and are slow as well as expensive for the client user. An alternative solution is to use fully-homomorphic encryption (FHE) schemes that allow arbitrary computation on encrypted data [31]. Though promising, these techniques are presently too slow for practical usage [30].

As a step towards practicality, we investigate execution of user-level benign applications, using *partially homomorphic encryption schemes (PHE)*. Partially homomorphic encryption schemes allow specific (limited) computations on encrypted content [24, 48, 58]. As the main contribution of this paper, we develop a new compiler called AUTOCRYPT that takes real-world C applications and automatically transforms them to operate on encrypted data. The transformed (or AUTOCRYPT-ed) x86 code runs in the untrusted VM and is allowed to invoke only 6 fixed *hypervisor API*, by making a *hypercall*, to support useful functionality. Our design ensures that all computations in the untrusted VM are *privacy preserving*, as defined formally in Section 3, using standard definitions of privacy-preserving homomorphic computation [58, 64]. The trusted hypervisor approach combined with AUTOCRYPT opens up a compelling alternative to the 2 existing solutions for enabling rich functionality in an uncompromised web server setting. It provides a high-bandwidth channel between an untrusted VM and hypervisor (via hypercalls) and adds little to the hypervisor's TCB.

In designing AUTOCRYPT, we carefully avoid schemes such as fully-homomorphic encryption (FHE), which are presently too slow for practical usage [30, 31]. To avoid using FHE, the hypervisor APIs allow performing safe conversions between encryption schemes. AUTOCRYPT analyzes program written in C and internally infers an encryption type for each variable in the program. We design a secure type system that combines various partially homomorphic encryption (PHE) schemes to support a subset of C operations. This type inference captures the right homomorphic encryption to apply to the variable's content to enable the legitimate computation to work. To summarize, our solution consists of two parts: (a) a compiler AUTOCRYPT that type checks a given application and automatically transforms it to operate on encrypted data (b) a fixed set of static hypervisor APIs in the trusted hypervisor that switches data between encryption schemes.

Summary of Results. We test AUTOCRYPT on 30 Unix file processing applications from the COREUTILS package available on a typical LAMP stack, and 3 custom programs commonly used in image processing applications. AUTOCRYPT automatically transforms the 25 COREUTILS applications and all the custom applications to AUTOCRYPT-ed programs with *no* developer effort, showing the expressiveness of our new abstractions. Our experiments validate our three main empirical hypotheses that (a) only a small fraction of internal program state and code needs transformation (but is difficult to manually identify) (b) several programs can use a combination of inexpensive PHE techniques rather than expensive FHE to implement their functionality (c) performance of operating on encrypted data using combination of PHE techniques can be better than the traditional 'download-and-compute' mechanism and exisiting fully-homomorphic encryption schemes. We find that the execution time of AUTOCRYPT-ed COREUTILS programs grows linearly with data size. Our (unoptimized) AUTOCRYPT-ed programs are faster for 19 utilities and slower by factor of 2 to 6 for 6 utilities as compared to the 'download-and-compute' mechanism and faster by orders of magnitude than previous reported performance of FHE techniques. Thus, AUTOCRYPT makes homomorphic computation on a trusted hypervisor a compelling alternative to the 'download-and-compute' and FHE mechanisms.

Contributions. In this work, we focus on the problem of retrofitting partially-homomorphic encryption (PHE) schemes transparently to existing application with no developer effort. To our knowledge, our work is one of the first to investigate how to automatically transform existing C applications using PHE schemes transparently without requiring any expertise or knowledge on behalf of the developer. Prior work has developed transformation tools for secure function evaluation on encrypted data, but mostly in the context of interactive secure two-party protocol implementations [32, 36–38, 43]. We target a new class of applications, motivated by our conceptual setting of a vulnerable web server hosted on a trusted hypervisor.

Specifically, we claim the following main contributions:

- We develop an automatic compilation tool for applications written in C language called AUTOCRYPT, which transparently embeds homomorphic computation to transform programs to operate on encrypted data.
- AUTOCRYPT provides experimental basis to the hypothesis that many applications can utilize a combination of PHE techniques instead of the more general (and expensive) FHE techniques. We successfully transform 25 (out of 30 studied) existing file-processing Unix utilities and 3 custom programs with no developer effort. This shows that the cheaper PHE techniques are sufficient for most of the applications we study, with a limited support from the trusted hypervisor.

2. OVERVIEW

To enable protection of sensitive content, we envision a conceptual architecture (Section 2.1) which splits the web server into 2 VMs, where the main web processing is an untrusted VM. The main focus of this paper is providing a small, secure hypervisor API and a type-based tool to transform existing applications to operate on PHE encrypted data.

2.1 Setting: A Split Server Architecture

Instead of trusting the web server VM completely, we envision a setting where the server is partitioned into 2 server VMs: the *main* untrusted server VM which handles the bulk of the client request processing and a *thin* authentication and key server. The thin authentication server handles the user authentication (via HTTPS) and initializes the hypervisor with the allowed encryption keys at the start of a session. The hypervisor then launches the untrusted main VM to handle the client's request in the session.

The untrusted VM executes a standard LAMP web stack which, in part, processes sensitive or encrypted files. Applications which implement the server's legitimate functionality are pre-transformed using our AUTOCRYPT on the client side. A client with his secret key and AUTOCRYPT tool transforms an application to AUTOCRYPT-ed program and uploads it to the web server.

In our design, files are stored encrypted under *more than one* partially homomorphic encryption schemes on the web server. For simplicity, we assume that each file has n ($= 3$) copies on the server, each encrypted with a different encryption scheme under the user's secret key set κ. Each encryption scheme allows a certain kind of computation to be homomorphically executed on the encrypted file contents, under inputs encrypted with κ provided by the trusted client. For example, a file encrypted using searchable encryption allows only the trusted client to search for constant keywords on a file content, without revealing the encryption keys to the server [58]. Similarly, homomorphic schemes with respect to addition / subtraction (e.g. Paillier [48]), multiplication (e.g. Elgamal [24]) allow respective computations on the data. Our main goal is to transparently transform a program f that isn't designed

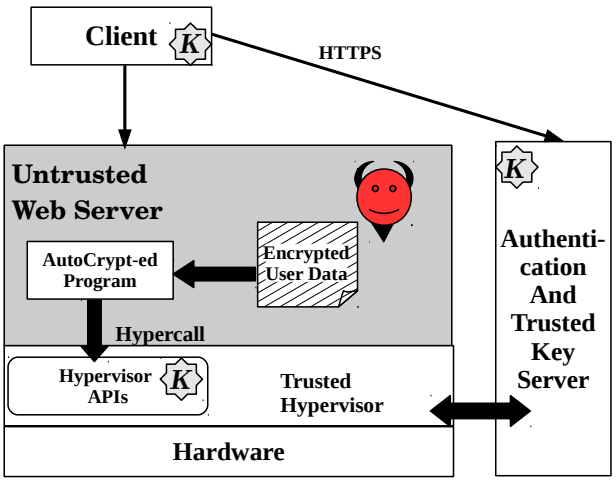

Figure 1: Trusted Hypervisor Setting. Secret key set K is present with the client, authentication server and the trusted hypervisor. The attacker can only see the encrypted data and the AUTOCRYPT-ed program.

to operate on encrypted data, into a program f' which can operate on encrypted sensitive data. The program f' takes encrypted inputs and executes to produce encrypted results. Supporting this scenario enables, for example, several standard Unix file processing utilities on encrypted files.

We trust the underlying hypervisor to be secure. Existing techniques to ensure its integrity can be used [45]. In this design, the hypervisor does not allow the untrusted VM to execute any unprivileged code dynamically in the privileged hypervisor mode (e.g. DOM0 in Xen). The hypervisor permits the untrusted VM to invoke a set of hypervisor APIs. These APIs implement a small set of static functionalities for decrypting data in one scheme and encrypting it in other scheme as detailed in Section 2.4.

2.2 Threat Model

We assume that the adversary can compromise and run arbitrary code in the untrusted VM during the session, as shown in Figure 1. We argue about the attacker's capability in these cases and design a robust solution to ensure confidentiality of sensitive web content.

Exfiltration Channels for Adversary. In this setting, the untrusted main VM only sees encrypted files and computed information from them. There are 3 channels it can utilize to exfiltrate data. First, it can directly send encrypted content to a remote attacker — however, such an attacker would not have the legitimate user's encryption keys, and so would not be able to decrypt the data. A second channel is to leak decrypted content via the client-side code. Defenses to prevent client-side data exfiltration have recently been proposed using features in HTML5 such as temporary origins [12]. We assume that with the help of the trusted authentication and key server, we can privilege separate the decryption and data export channels in a secure origin on the client-side. We do not discuss these defenses in this work. Finally, the attacker can learn the sensitive information using the hypervisor API exposed to the unprivileged code executing in the main VM. This threat is the main focus of our techniques. We show that the hypervisor's runtime service API cannot be used to leak information about computed values (see Section 3.4). That is, the untrusted VM cannot perform undue computation on sensitive data to leak information.

Ensuring Confidentiality On a Trusted Hypervisor. In this setting, however, ensuring confidentiality when a known computation is performed on the data files is an important challenge. We assume that the attacker can infer which application is being executed, i.e. we do not require functional privacy. We conservatively model the attacker's *knowledge set* to include f, f', the encrypted inputs and outputs of f', and everything that can be computed[1] from these. The installed malware on the untrusted VM can independently run AUTOCRYPT-ed f' on available encrypted data, and can observe each memory access and instruction executed in f'. Therefore, any information implied by the path constraints along any given execution are assumed to be part of the attacker's knowledge set. In addition, the adversary is not restricted to running only benign f' on the encrypted data available to it. Indeed, for a data element encrypted under κ with a particular homomorphic scheme (note that κ is a key set), the malware can run arbitrary computation permissible by the scheme. The catch, of course, is that it is bound to compute with data which is already encrypted under κ with the same homomorphic encryption scheme. The results are decrypted outside the VM and not by the attacker (as the key is unknown to the attacker). Therefore, an important challenge is to limit the expressiveness (or richness) of the computation possible on sensitive data in the untrusted VM, while allowing "sufficiently interesting" benign functionality to execute.

2.3 Problem Definition

Our goal is to develop AUTOCRYPT, a tool that given a benign application f, checks if its computation can be transformed into a safe computation to run on the server. If so, AUTOCRYPT transform f into f' while preserving the semantics / correctness of the original computation and guaranteeing the safety of f' as per the definitions of privacy preserving execution.

More precisely, let an application f be a 3 tuple consisting of (a) a set of valid instruction traces S, operating on sensitive inputs $\vec{I}$, a set of public program constants C to produce sensitive outputs. Let Σ be the input alphabet and $\mathbb{W}$ represent the publicly known program constants encrypted under search scheme. We seek a transformation procedure $AutoCrypt : (S, \vec{I}, C) \mapsto (S', \vec{I'}, C')$, such that it satisfies the following three properties.

- *Correctness :* The transformed program f' preserves the semantics of f, i.e. $\forall i \in \vec{I}, S(i) = D_\kappa(S'(E_\kappa(i)))$.
- *Privacy-preserving execution:* The execution of f' under sensitive inputs $\vec{I}$ is privacy-preserving. We provide a precise definition of security achieved by our scheme as *reduced indistinguishability* (see below & Section 3.4) which is weaker than the strict computational non-interference, but admits a large variety of interesting web server functionality.
- *Minimal Control Flow Leakage:* The number of encrypted program constants under search scheme $\mathbb{W} \subseteq C$ is minimal, i.e., consisting only those constant expressions that are used in conditional operation in the original f.

Security Property: Privacy-preserving execution modulo $\mathbb{W}$. We aim to ensure that all executions in the untrusted VM, including f' and the adversarial computation, are privacy-preserving computation (modulo the parameter $\mathbb{W}$ necessary to support benign applications). Our goal is to enable interesting benign functionality in the untrusted VM, without giving a malicious computation the capability to leak sensitive information beyond what is implied by its

[1]according the standard definitions of computationally bounded adversaries

knowledge set. For *all* computation in the untrusted VM, including the AUTOCRYPT-ed program f', computation on encrypted data is restricted only to homomorphic operations. In addition, computation in the VM can make a small set of hypercalls. In our setting, the amount of information leakage in the adversarial computation is lower bounded by leakage permitted in benign functionality. In permitting benign executions, on one end, we could enforce a strong non-interference property (in the computational sense [28]) on *all* computation in the VM. However, many practical programs we study leak some information, via control flow channels, but the information leaked is small. We aim to admit a limited set of such benign functionalities, but bound the adversarial computations simultaneously. In essence, we aim to prevent computation (including the adversary's computation) from using dynamically computed values and checking them in conditional operations — this does not preclude computation using static encrypted values being used in a small set of permitted conditional checks.

To this end, we enforce a confidentiality property called reduced indistinguishability, which is parameterized by a set $\mathbb{W}$, on all computation in the untrusted VM. Intuitively, $\mathbb{W}$ captures the subset of the input alphabet Σ which is permitted to leak (and necessary to support benign functionality). For all input $\Sigma - \mathbb{W}$, reduced indistinguishability offers IND-CPA security. We define reduced indistinguishability along the lines of IND-CPA more formally in Section 3.4. We term all executions which satisfy reduced indistinguishability as *safe*. In practice, we find that the $\mathbb{W}$ is a small set — for supporting all our 25 case studies (Section 5), $|\mathbb{W}| = 12$ (where $|\Sigma| = 256$).

Scope. In this work, we focus on preserving the *confidentiality* of the encrypted inputs and outputs, not the integrity or freshness of the outputs of the executed program f'. Cryptographic techniques to handle integrity and freshness are being investigated, but are beyond the goals of this work [17, 29]. In our threat model, we assume that the legacy application is not designed to conceal lengths of its inputs / outputs; our encryption techniques are assumed to be length-revealing. Of course, techniques can be used to mitigate the last assumption [40], which are orthogonal to our techniques. We assume that the underlying homomorphic schemes are semantically secure, and their implementations blind side-channels using known techniques [20]. We do not aim to conceal information leakage through memory access pattern, which can be achieved using oblivious RAM techniques [59].

In this work, we focus on applications that read sensitive files and produce HTTP outputs decrypted on the client browser. For example, web servers need to perform content sniffing on files types (images, PDFs, etc.) to generate the appropriate HTTP MIME type header. Other examples include standard Unix utilities to compute statistics (e.g. word count, etc.) for web files, keyword search, checksum computation, base64 encoding, which are useful especially in cloud-hosted source code repositories (e.g. Github, Bitbucket). We do not support applications, outputs of which are to be stored on the web server itself and further processed by AUTOCRYPT-ed programs. Our evaluation is a preliminary first step in this direction; future research is needed to improve both the expressiveness and performance of AUTOCRYPT-ed programs.

2.4 Solution Overview

We design a new compiler—AUTOCRYPT that takes existing applications in `C` and transforms them automatically to work on encrypted data without any developer effort. In our design we combine various PHE schemes to enable a subset of operations that a general programming language supports. We develop a type system that type checks any application before AUTOCRYPT transforms it to AUTOCRYPT-ed program. An application which does not type checks according to our system is considered 'unsafe' for transformation. AUTOCRYPT identifies the sensitive variables in a well-typed application and infers the right encryption type for its content and transforms it to work on encrypted inputs.

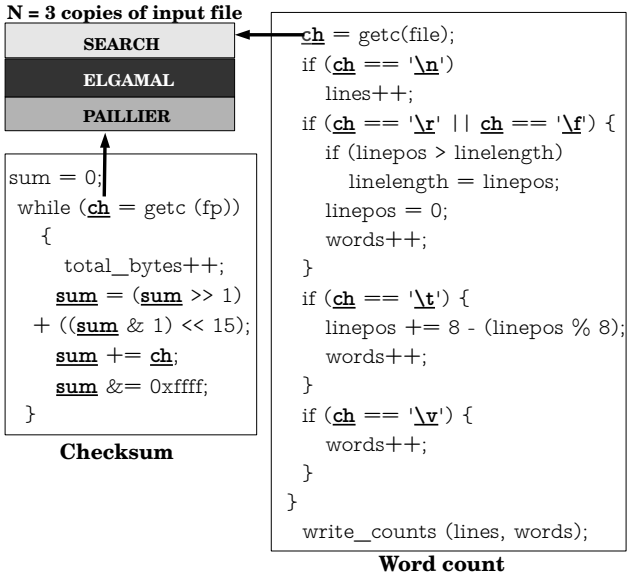

Figure 2: Program Layout of AUTOCRYPT-ed Programs. The variable `ch` is the sensitive variable and is assigned encryption type as Paillier in `checksum` and search type in `word count` program.

The input file to the transformed program is pre-encrypted with $n(= 3)$ different homomorphic encryption schemes. A transformed application operates on the correct copy of an input file depending on the encryption types inferred for the sensitive variables in the program. We explain the program layout of two transformed applications that use one of 3 copies of the input file in Figure 2. The `checksum` program that performs mathematical operations fetches the sensitive input from the Paillier encrypted copy of the input file whereas the `word count` program fetches its input from the search encrypted copy of the input file. For both the programs, `word count` and `checksum`, AUTOCRYPT identifies the variable `ch` as sensitive and infers an encryption type for it. In the `word count` program, `ch` is of search encryption type whereas in the `checksum` program, it has Paillier encryption type. The 5 constants (shown as boldface) in the `word count` program are also inferred the search encryption type, thus here $|\mathbb{W}| = 5$. AUTOCRYPT pre-encrypts them in the search encryption scheme during transformation of the program.

Hypervisor APIs. The `checksum` program shown in Figure 2 requires more than one homomorphic properties in the same program. Thus, our system infers different encryption types to the same variable depending on the operation in which it is used. At any given time, a program variable is encrypted in either search encryption (`srch`), Paillier encryption of bits of integer ($\texttt{pal}_1$), Paillier encryption of integer ($\texttt{pal}_8$) or Elgamal (`egml`) encryption scheme depending on the operations performed on it. One possible approach is to ask the client to decrypt the data in one encryption scheme and encrypt it in another scheme. But the network latency between the client and the server degrades the performance for applications like `checksum` where conversion between the encryption schemes is required for every character. To allow switch-

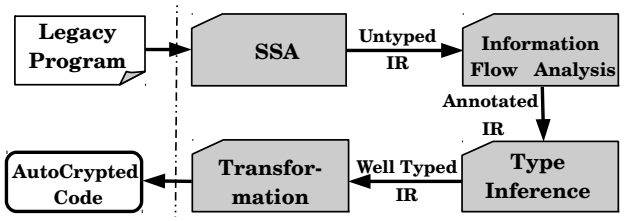

Figure 3: Design overview of AUTOCRYPT.

ing between the encryption schemes while still executing the program with acceptable performance, we provide a fixed set of static hypercalls in the trusted hypervisor. The hypervisor APIs support only 6 fixed functionalities to switch the sensitive data between different homomorphic encryption schemes. The hypervisor does not execute any unprivileged arbitrary code on the sensitive data, thus our increase in the hypervisor TCB only consists of the following 6 hypercalls for conversion between encryption schemes.

- $\texttt{pal}_8$ to $\texttt{pal}_1$
- `egml` to $\texttt{pal}_1$
- `egml` to $\texttt{pal}_8$
- $\texttt{pal}_1$ to `egml`
- $\texttt{pal}_8$ to `egml`
- static memory lookup

Thus, using the three encrypted copies of the sensitive input and the hypervisor APIs, AUTOCRYPT transforms majority of the existing applications to operate on encrypted data.

3. AUTOCRYPT DESIGN

In this section we describe the end-to-end design of AUTOCRYPT and discuss security guarantees of our system.

3.1 Overview

AUTOCRYPT takes as input a legacy application that operates on unencrypted files and transforms it to an AUTOCRYPT-ed program which operates on encrypted files and command-line inputs (as shown in Figure 3). The developer annotates all reads from sensitive file APIs and command line arguments in the initial program with `high` information labels. AUTOCRYPT takes the annotated program and performs the following steps:

Conversion to SSA form. In the first step, AUTOCRYPT converts the input program to an intermediate representation in Static Single Assignment form (SSA). SSA form splits the lifetime of a program variable into many smaller ones by assigning a new version every time it is re-defined. This allows AUTOCRYPT to separate out different operations over the lifetime of a variable, thereby increasing the opportunity to separate out operations that are incompatible under a single homomorphic encryption scheme. Every operation is represented as a separate instruction in the Internal Representation (IR). We refer to the output of this step as the *untyped IR*.

Information Flow Analysis. AUTOCRYPT then analyzes the untyped IR for variables and constants that need encryption. In this analysis, AUTOCRYPT tracks and propagates direct data dependencies of program variables on `high`-labeled inputs, file inputs and the outputs of the application. Our static data-dependency tracking is flow-sensitive. This analysis is akin to the standard variable-level taint-tracking [27]. Information flow analysis produces an annotated IR, in which all the operands are marked as either `high` or `low`.

Type Inference. AUTOCRYPT infers the precise homomorphic encryption scheme to be used for each `high` operand in the annotated IR. At the end of this step, it labels each operation with an encryption type that signifies its associated encryption scheme. To perform this step, AUTOCRYPT uses the standard type inference techniques. The output of this step is a well-typed IR — that is, it satisfies a formal set of type rules we define in Section 3.3.

Transformation. Finally, AUTOCRYPT transforms the well-typed IR into target x86 code using a syntax-directed translation step. In this step, each original computation is converted into the corresponding homomorphic computation. In addition, this step handles various features of real-world C programs, such as encoding into machine words, handling pointers, structures, C libraries, and so on. We detail these in Section 4.

3.2 Background on Encryption Schemes

We briefly introduce the partially homomorphic schemes used in AUTOCRYPT.

Searchable Encryption (`srch`). Searchable encryption scheme allows the owner to outsource encrypted data while maintaining the selectively-search capability over it. Keyword based symmetric searchable encryption enables such remote searching on encrypted data by an untrusted server [58]. Specifically, the untrusted server uses an encrypted token (unique to every search keyword) to determine whether a document contains the word, without learning anything else.

AUTOCRYPT uses this encryption scheme to support operators such as equal to. Previous works discuss various schemes to support similar operations [16, 58]. Among these, AUTOCRYPT uses CryptDB implementation of searchable encryption [51], based on the cryptographic protocol of Song et al. [58]. In AUTOCRYPT, the search keyword is mostly a character and hence both the keywords and file encryption are at character level as opposed to word level encryption in CryptDB. Note that the deterministic encryption schemes also support such equality checks. But it is a weaker encryption scheme revealing information about the underlying plaintext. The search scheme AUTOCRYPT uses is randomized and is thus CPA secure [58]. Hence, an attacker is bound to search only those terms which are available in an encrypted form either as program constants or as inputs and outputs for client queries.

Paillier Encryption. AUTOCRYPT uses the encryption scheme proposed by Paillier [48], to support various arithmetic and bitwise operations on encrypted data. Since the operations are homomorphic, the results are also encrypted under Paillier scheme. The encryption scheme is semantically secure and assures guarantees equivalent to randomized encryption. AUTOCRYPT classifies Paillier scheme into two categories.

i. Paillier encryption of an integer ($\texttt{pal}_8$): Integer value of a `high` variable is encrypted using Paillier encryption. This encryption is necessary for arithmetic operations such as addition, which operate on the integer representation.

ii. Paillier encryption of the bits of an integer ($\texttt{pal}_1$): To handle intermediate bitwise computations such as XOR, that rely in an essential way on the binary representations of their input values, AUTOCRYPT encrypts every bit in the binary representation of an integer.

Elgamal Encryption (egml). Elgamal encryption scheme is used to homomorphically multiply two encrypted operands such that the result is also encrypted under Elgamal scheme [24]. AUTOCRYPT demonstrates the use of this scheme for homomorphic matrix multiplication in Section 5. Support for regular expression matching and DFA can also be added to AUTOCRYPT by using this encryption scheme [62].

3.3 AUTOCRYPT Type Inference

Although standard information flow rules assist to infer all the program entities that are `high` [53, 57], inferring the corresponding encryption scheme is a non-trivial task. There are several subtle factors such as control flow, program constants, lookups using encrypted indexes, and complex arithmetic operations requiring conversion from one encryption type to other. These challenges must be carefully addressed so as to preserve the security guarantees of a legacy application. To this end, we present our AUTOCRYPT framework armed with a formal type system which checks whether a `C` application can be transformed to a *safe* application.

Main insight. An annotated IR has all operands marked `high` or `low`. Our type system allows arbitrary computation in expressions computed from `low` values, and these resulting values are not encrypted (either statically or at runtime). The adversary, thus, cannot use such `low`-labeled values to compute or check any encrypted data.

For `high`-labeled variables, we permit them to either be used in conditional checks or in arithmetic / bitwise operations, but not both. Specifically, `high`-labeled values can only be used in equality / inequality checks with other operands, and our type system forces them both to be encrypted under the search scheme. In effect, this prevents all values used in conditional checks from being used in *any* arithmetic / bitwise (non-conditional operations) as they are in different encrypted forms at runtime.

Remaining `high` values can be computed in arithmetic / bitwise schemes, but these are never permitted to be converted to search-encrypted values. This is checked by our static type rules, and no hypercall allows to convert between searchable encryption and other homomorphic schemes at runtime. Together, this enables a conceptual partitioning of encrypted memory between searchable and non-searchable values.

Language. Figure 4 shows the syntax for a simple typed language supported by AUTOCRYPT. For brevity, we consider this simple language to illustrate our approach. However we evaluate on various `C` programs as discussed in Section 5. The simple language is a subset of `C` and is expressive enough to model many programs. Our type-based approach is general and can be enhanced further to incorporate richer language features. We discuss the subset of `C` constructs we support in Section 4. The language has two possible information-flow labels `low` and `high`. Program variables and constants are also classified based on the required encryption types viz. no encryption i.e. plain text (`ptxt`), search encryption (`srch`), Paillier encryption ($\texttt{pal}_8$, $\texttt{pal}_1$) and Elgamal encryption (`egml`).

Operations that can be computed with the same encryption scheme are syntactically combined in the language. For bitwise operations in $\odot$ and $\circledcirc$, one or both the operands must be in bitwise Paillier encryption scheme ($\texttt{pal}_1$). Arithmetic operations in $\oslash$ need operands encrypted in integer Paillier scheme($\texttt{pal}_8$). For multiplying two encrypted variables ($*$), they must be encrypted under Elgamal scheme (`egml`). In conditional operations ($\otimes$), an encrypted `high` variable can be compared against encrypted public program constant. Hence, both its operands must be encrypted under search encryption scheme (`srch`). Note that known program constants in our language are typed to be either encrypted or non-encrypted but they are publically known and hence always `low`.

Type System. Each expression has a type τ, which is a tuple of two labels: information-flow label (`low` and `high`) and encryption type label. We write $\Gamma \vdash e : \tau$ to denote that the expression e has type τ according to the typing rules in Γ typing environment, where τ denotes type of e. Such an assertion is called a typing judgment.

Types	τ	::=	(Q, T)
Type Qualifiers	Q	::=	$high \mid low$
Base Types	T	::=	$\alpha \mid \beta$
Plaintext	α	::=	`ptxt`
Encrypted	β	::=	$\beta_1 \mid \beta_2$
	β_1	::=	`srch`
	β_2	::=	$\texttt{pal}_8 \mid \texttt{pal}_1 \mid$ `egml`
Expressions	e	::=	e1 $\odot$ e2 | e1 $\circledcirc$ e2 | e1 $\oslash$ e2 | e1 $*$ e2 | lval | `const` (i:τ)
	lval	::=	`var` (v:τ)
Bitwise	$\odot$	::=	& | ^ | |
	$\circledcirc$	::=	<< | >>
Arithmetic	$\oslash$	::=	+ | -
Boolean	b	::=	bool | e1 $\otimes$ e2 | b1 $\ominus$ b2
	bool	::=	`true` | `false`
Conditional	$\otimes$	::=	== | !=
Logical	$\ominus$	::=	AND | OR
Commands	P	::=	lval := e | lval := ARR[var] | if b then P else P' | while (b) do P | `return` r | f(e_1, e_2, . . ., e_n) {P;}

Figure 4: The grammar of a simple subset of `C` language supported by AUTOCRYPT.

Typing rules for Expressions. The expression typing judgement $\Gamma \vdash e : \tau$ states that at a given program location, the expression e has a type τ. The Γ denotes the type environment that maps variables to the target language base types.

Typing rules for Commands. The type system is flow-sensitive because it permits the mappings of variables to types in the typing environment Γ to change from one program location to another. Typing judgements for the commands in Figure 5 capture the effects of execution of the command on the type environments and have the form $\Gamma \vdash c \Longrightarrow \Gamma'$. The type environment Γ is changed to Γ' as a result of the execution of the command.

Type Inference. We infer flow-sensitive types based on our type rules in Figure 5 using standard techniques proposed in previous work [27]. We omit the type rule for instructions taking all operands with `low` labels.

The rules for inferring encryption types are as follows.

- `SRCH` : If one operand to a conditional operation ($\otimes$) is `high` searchable variable and other is a program constant, then both the operands are typed to `srch` and the result is a `low` boolean value.
- $\texttt{PAL}_8$`-A & B` : For mathematical operations in $\oslash$, one or both operands can be `high`. For $*$ one operand is high and other is low. Then, the `high` operands have type $\texttt{pal}_8$ and the result of computation is also $\texttt{pal}_8$ type.
- $\texttt{PAL}_1$`-A & B` : For bitwise operations in $\odot$ and $\circledcirc$, the `high` operand is given type $\texttt{pal}_1$ while the other operand can be `low` or `high`. The resulting value is also in $\texttt{pal}_1$.
- `EGML` : If both the operands to a $*$ operation are `high`, then the operands and result have `egml` type.
- The encryption type to $lvalue$ for statements is specified in the type rules.

Type Conversion. Figure 6 depicts the type lattice enforced by AUTOCRYPT type system. Type conversions are non-descending i.e. conversions only go upwards or are at the same level [23]. Our type system only supports the conversions for which there is a corresponding runtime hypervisor API. We explain the safe up-casting conversions below, and give examples of the unsafe down-casting operations that are not supported by our hypervisor at runtime.

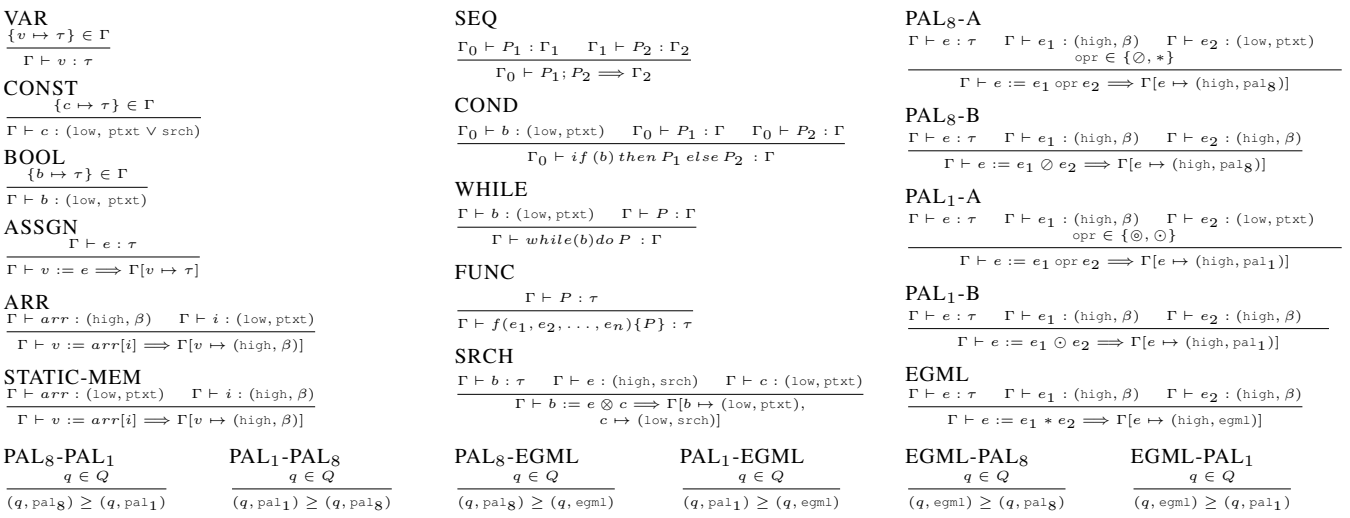

Figure 5: Type Rules for expressions and commands in AUTOCRYPT. Γ is the typing environment that maps variables and constants to type qualifiers `high` or `low` at each program locations. It also maps the encryption types viz. `ptxt, srch, pal`$_8$`, pal`$_1$ `or egml` to each variable. Type qualifiers are flow sensitive. We omit the type rule for instructions taking all operands with `low` labels here for brevity.

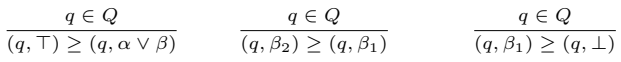

Figure 6: Type Lattice rules for AUTOCRYPT

(a) Safe conversions (Up-casting):
Type rule PAL$_8$−PAL$_1$ and PAL$_1$−PAL$_8$ allow interconversion of `high` variable encrypted under Paillier integer and Paillier bit scheme. Similarly, PAL$_8$−EGML, PAL$_1$−EGML enables conversion from Paillier (both integer and bit) encryption to Elgamal scheme and EGML−PAL$_8$, EGML−PAL$_1$ from Paillier to Elgamal scheme. Up-casting from searchable encryption scheme to Elgamal and Paillier schemes is safe according to the type-lattice.

(b) Unsafe conversions (Down-casting):
AUTOCRYPT type rules do not allow conversion from Elgamal or Paillier encryption scheme to searchable encryption scheme. If such conversions are allowed by the type system, an adversary can build a program which arbitrarily converts the results of arithmetic homomorphic operation say $x = y+10$ to searchable cipher text. Then, he can compare x to the available program constants encrypted as searchable tokens. By such a series of instructions, an adversary can predict the initial value of y with a non-negligible advantage, and hence win the indistinguishability game. Thus, to prevent such attacks, the conversion to searchable encryption by down-casting is not allowed in AUTOCRYPT.

All static program constants are marked `low` as these are publically known. Program constants used in the relational operations with `high` operands need to be encrypted. All the other program constants need not be encrypted, as they are used directly in the transformed homomorphic operations (see Section 3.5). The type rules for the remaining expressions are straight forward, except the rule to handle lookup in a static memory using encrypted indexes.

Encrypted indexes to static memory. A lookup in a statically initialized array of public constants using a `high` index can reveal the encrypted index, if the adversary can observe the lookup result. Our type system handles this securely (rule STATIC−MEM) by forcing the outputs of such memory lookups to be `high` and encrypted under the same scheme as the encrypted index. To support this operation at runtime, our hypervisor provides an hypervisor API that performs the lookup obliviously. Specifically, the hypercall for `ARR[i]` lookup operation ensures that $\forall\, m \in\ RANGE(arr)$, $(E(arr[D(i)]), E(m))$ are computationally indistinguishable by the adversary.

We point out that, in principle such lookups could be supported by using homomorphic evaluation of polynomials [62]. However, for $\Sigma = 256$ the polynomial degree is high and the evaluation could be slow. Alternatively, AUTOCRYPT's hypervisor API supports static memory lookup as follows: All statically initialized lookup arrays are saved as plaintext in the trusted hypervisor. Every lookup from such arrays using an encrypted index is converted to a hypercall. In the trusted hypervisor environment, the service API first decrypts the index and fetches the corresponding plaintext from the static (unencrypted) memory. The plaintext is then encrypted under the same randomized encryption scheme as the index, using a fresh random nonce.

3.4 Security Guarantees

AUTOCRYPT does not rely on the type system discussed in Section 3.3 for security. The type system only assists in checking program's compatibility for AUTOCRYPT conversion and for inferring encryption types of program variables. However, even bad program (ill-typed) can execute on the malicious web server. To this end, our design of the hypervisor APIs together with the use of homomorphic encryption schemes safeguard against such ill-typed programs, while ensuring that for all executed code in the VM (including adversarial and benign code), has a specific security property which we define below.

Privacy-preserving Executions. Any executing program in the VM can be viewed as a sequence of selective statements from the program depending on the input. A program trace corresponds to one such execution sequence. More formally, we define:

DEFINITION 1. ***(Program Trace** T**)** The program trace $T : \vec{I} \rightarrow \vec{O}$ is a sequence of statements $\langle S_1, \ldots, S_n \rangle$ operating on encrypted input $\vec{I}$ yielding encrypted output $\vec{O}$ in deterministic program P.*

As previously mentioned, the program can only use a small set of publicly known program constants $\mathbb{W}$ in conditional comparisons (control flow decisions), made available in the VM for supporting some benign applications. We define $\mathbb{W}$ and a program P formally as below.

DEFINITION 2. ***(Encrypted Search Constants $\mathbb{W}$)*** *For a program P, $\mathbb{W} = \{W_1, \ldots, W_n\}$ such that $\mathbb{W} \subseteq \Sigma$ and W_i is a publically known program constant encrypted under search scheme.*

W is a subset of possible input alphabets Σ. Specifically, all the publically known program constants which are marked as `low` and are also encrypted under search scheme belong to the set $\mathbb{W}$.

DEFINITION 3. ***(Program P)*** *A Program $P = (\mathbb{T}, \mathbb{W})$ where $\mathbb{T} = \{T_1, \ldots, T_n\}$, and $\mathbb{W}$ are encrypted search constants.*

The execution of a program trace interleaves between the execution in the untrusted VM and in the trusted hypervisor. Therefore, we can define the projection of a trace visible to an entity (either the adversary or the hypervisor) as follows: P may have other program constants; but we ignore them in our definitions for clarity as they are unimportant for our security reasoning.

DEFINITION 4. ***(Projection P_i)*** *Projection Π of a trace T onto E, written as $\Pi_E(T)$, is a subsequence of a program trace T visible to an entity E.*

Given a program trace, an entity can view only some statements in the execution trace. For example, the untrusted VM (say A) cannot see the internal hypervisor implementations of the hypercalls and encrypted memory lookups in an AUTOCRYPT-ed program. Thus, these statements will not be a part of the $\Pi_A(T)$. The following 2 security definitions constitute the confidentiality of the sensitive inputs, apart from the values which the adversary can directly compare with $\mathbb{W}$ by using equality checks on search encrypted bytes.

DEFINITION 5. ***(Input Reduction Function R)*** *For an input vector $\vec{I}$ for a given program P, we define the input reduction function $R(\vec{I}, \mathbb{W})$ which reduces $\vec{I}$ by discarding all the W_j values in $\vec{I}$ at positions where $\forall\, W_i \in \mathbb{W}, W_j = W_i$.*

DEFINITION 6. ***(Reduced Indistinguishability modulo $\mathbb{W}$)*** *For a given trace $T : \vec{I_i} \rightarrow \vec{O_i}$ and $\mathbb{W}$, we say that T has reduced indistinguishability modulo $\mathbb{W}$, iff $\forall\, m_0, m_1 \in R(\vec{I}, \mathbb{W})$, the advantage $Adv(A) = |Pr[\Pi_A(T_{m_0}) = 1] - Pr[\Pi_A(T_{m_1}) = 1]| < \varepsilon$ for all probabilistic polynomial time-bounded adversaries A, where ε is negligible function $negl(n)$.*

Thus, along a given trace T_i in the program P, the advantage of all adversaries is negligible and it cannot distinguish two inputs from the reduced input vector. This reduces to standard definitions of ciphertext indistinguishability under IND-CPA when $\mathbb{W} = 0$.

DEFINITION 7. ***IND-CPA for Search Encryption***. *For a given trace $T : \vec{I_i} \rightarrow \vec{O_i}$ and a set of encrypted search constants $\mathbb{W}$ available to the adversary, if $\vec{I_i}$ is search encrypted, then $\forall\, m_0, m_1 \in R(\vec{I}, \mathbb{W})$, the advantage $Adv(A) = |Pr[\Pi_A(T_{m_0}) = 1] - Pr[\Pi_A(T_{m_1}) = 1]| < \varepsilon$ for all probabilistic polynomial time-bounded adversaries A, where ε is negligible function $negl(n)$.*

The above definition assumes the server has only the searchable encryption copy of the user's file. The proof of indistinguishability under IND-CPA for search encryption follows directly from previous work [58].

DEFINITION 8. ***IND-CPA for Non-Search Encryption***. *For Trace $T_i : \vec{I_i} \rightarrow \vec{O_i}$, $\forall\, m_0, m_1 \in \vec{I}$, the advantage $Adv(A) = |Pr[\Pi_A(T(_{m_0}) = 1] - Pr[\Pi_A(T_{m_1}) = 1]| < \varepsilon$ for all probabilistic polynomial time-bounded adversaries A, where ε is negligible function $negl(n)$.*

If we assume that there is only non-search encrypted copy of the file on the server, $\mathbb{W} = 0$ and hence $R(\vec{I}, \mathbb{W}) \equiv \vec{I}$. Thus, for a purely non-search encryption the indistinguishability property holds. The proof follows directly from the IND-CPA security of individual homomorphic schemes [64].

Definitions 6, 7 and 8 are independently true only when a corresponding single copy of encrypted file(encrypted either under searchable encryption or non-searchable encryption scheme) is available. If both copies are present simultaneously on the server, logically the definitions holds for $min(I, R(\vec{I}, \mathbb{W}))$ i.e. on the reduced input $R(\vec{I}, \mathbb{W})$.

The following theorem states that interconversion between non-search schemes preserves indistinguishability under IND-CPA.

LEMMA 1. ***IND-CPA for non-search Composition***. *For a safe program P, if the functions $f_1, \ldots, f_n$ are individually secure non-search homomorphic functions under IND-CPA, then the composition $(f_i \circ f_j)(x)$ where $i, j \in \{1, \ldots, n\}$ satisfies reduced indistinguishability.*

Our conversions between the non-search encryption schemes can be proved using bisimulation induction on a type system proposed in recent work [28].

Our final theorem says that all execution traces in the hypervisor are privacy preserving, i.e. follow reduced indistinguishability for a given $\mathbb{W}$. This stems from a critical design choice: the hypervisor does not permit converting from non-search to search types during execution. Let $NS(T)$ and $S(T)$ be set of runtime values encrypted under non-search and search encryption in T respectively. Our hypervisor ensures that $NS(T) \cap S(T) \subseteq \mathbb{W}$. This establishes the following safety condition for any program running in the VM:

THEOREM 1. ***IND-CPA of all VM Executions***. *For a program $P = (\mathbb{T}, \mathbb{W})$, executing on our hypervisor, reduced indistinguishability modulo $\mathbb{W}$ holds if $\forall\, T \in \mathbb{T}$, $NS(T) \cap S(T) \subseteq \mathbb{W}$.*

PROOF SKETCH. Proof follows from Definition 7 and Lemma 1. We sketch the outline below: $\forall\, v_{ns} \in NS(T)$ and $\forall\, v_s \in S(T)$ hypervisor does not permit conversion at runtime from values v_ns to corresponding v_s. Thus, the only values v for which both v_s and v_{ns} are known are in set of encrypted search constants used in conditional operations. For P, $\mathbb{W}$ is a set of all such encrypted constants. From Definition 7, reduced indistinguishability holds for the traces comprising encrypted search constants. For non-search encryption, reduced indistinguishability follows from Lemma 1. ■

Well-Typed Programs. For benign programs, AUTOCRYPT aims to identify if all the traces in it can be made privacy-preserving. The type rules check this, and if well-typed, programs are transformed. We show that well-typed program indeed have reduced indistinguishability modulo $\mathbb{W}$.

THEOREM 2. ***Well-Typed Implies IND-CPA***. *If P is a well-typed program, then $\forall\, T \in P$, the reduced trace indistinguishability property holds for T.*

We omit a rigorous proof in this paper.

3.5 Transformation

Converting operations to homomorphic computation. We use CryptDB implementation of the search and Pailler encryption schemes and libgcrypt library for the Elgamal encryption scheme [3, 51]. Combining partially homomorphic properties of Paillier, Elgamal and search scheme, we transform various operations to operate on encrypted data. We give details of subset of `C` operations that AUTOCRYPT supports and the encryption type required by them in Table 1. Operations not listed in Table 1 are either not possible using homomorphic encryption schemes or are not safe and hence are not allowed by AUTOCRYPT.

Operation	Op1 Type	Op2 Type	Output Type
$add(+)$	$\texttt{pal}_8$	ptxt	$\texttt{pal}_8$
	$\texttt{pal}_8$	$\texttt{pal}_8$	$\texttt{pal}_8$
$mul(*)$	$\texttt{pal}_8$	ptxt	$\texttt{pal}_8$
	egml	egml	egml
$sub(-)$	$\texttt{pal}_8$	ptxt	$\texttt{pal}_8$
	$\texttt{pal}_8$	$\texttt{pal}_8$	$\texttt{pal}_8$
shl ($\ll$)	$\texttt{pal}_1$	ptxt	$\texttt{pal}_8$
shr ($\gg$)	$\texttt{pal}_1$	ptxt	$\texttt{pal}_8$
or ($\mid$)	$\texttt{pal}_1$	ptxt	$\texttt{pal}_1$
	$\mathbf{pal}_1$	$\mathbf{pal}_1$	$\mathbf{pal}_1$
and (&)	$\texttt{pal}_1$	ptxt	$\texttt{pal}_1$
	$\mathbf{pal}_1$	$\mathbf{pal}_1$	$\mathbf{pal}_1$
xor ($\hat{}$)	$\texttt{pal}_1$	ptxt	$\texttt{pal}_1$
	$\mathbf{pal}_1$	$\mathbf{pal}_1$	$\mathbf{pal}_1$
$equal$ ($==$)	srch	keyword	true / false
not_equal ($!=$)	srch	keyword	true / false

Table 1: Homomorphic operations supported by AUTOCRYPT. Column 2, 3 and 4 give the encrytion type for expressions of the form `Output = Op1 operation Op2`. The implementation of the operations or, and and xor comprise of hypercalls (shown in bold faced) when both its operands are encrypted. $equal$ and not_equal operations take one operand as a `keyword` which is a searchable program constant.

Encryption type conversion. Operations involving different encryption types may require conversion of intermediate results to other encryption schemes, for example converting an integer encrypted under Paillier encryption to Elgamal encryption. AUTOCRYPT inserts hypercalls at appropriate program locations for permitted convert operations. The hypervisor API in turn implements these operations. It first decrypts the input value using appropriate keys, and then re-encrypts this value using the keys for target encryption scheme. At last, it responds to the hypercall by returning this re-encrypted value to the untrusted VM.

4. IMPLEMENTATION

We implement AUTOCRYPT to automatically transform applications written in `C` to operate on encrypted data. In this section, we discuss the details of our implementation.

Analysis Pass. The sensitivity analysis phase of AUTOCRYPT is implemented as an analysis pass in the LLVM 3.2 infrastructure [42]. This phase comprises of conversion from LLVM IR to pure SSA form using LLVM `mem2reg` pass followed by following analysis.

The SSA form is then passed to the points-to analysis module which uses algorithm proposed by Hardekopf et al. [35]. All the virtual registers are grouped in alias sets and the information is maintained as pragmas to be used by the consecutive phases of AUTOCRYPT.

All user inputs and operations on data fetched from the file are marked as sensitive. This phase propagates the sensitivity of input by marking the appropriate variables and constants as sensitive and the records corresponding instructions as sensitive. AUTOCRYPT analyses all the functions of the program to collect a set of sensitive elements and instructions and logs the results in the form of Datalog facts.

After identifying all the possible sensitive constants and variables, AUTOCRYPT's inference module assigns them a type of encryption scheme using the type rules discussed in Section 3. AUTOCRYPT implements this phase in Datalog, which takes the list of sensitive variables, constants and corresponding instructions as facts [1, 11]. The typing rules discussed in Section 3 are given as constraints to Datalog which infers the encryption type for all the sensitive elements.The inference results of Datalog are fed back to the LLVM IR in the form of pragmas.

Category of file operations	Utilities
Basic operations	shred, truncate
Summarizing files	wc, sum, cksum
Formatting file contents	fmt, old, pr
Operating on characters	expand, unexpand, tr
Operating on fields	cut, paste, join
Operating on sorted files	shuf, uniq, comm, ptx, sort
Output of parts of files	head, tail, split, csplit
Output of entire files	cat, tac, nl, base64, od

Table 3: Classification of 8 categories of COREUTILS used for case study

Transformation Pass. AUTOCRYPT transforms the typed IR code to AUTOCRYPT-ed code. This phase is implemented as a transform pass in LLVM and it involves:

(a) computing the encrypted constants pool required by the program using the client's key
(b) instrumentation to load the program variables with correct values of encrypted data at run time
(c) adding calls to partially homomorphic functions which operate on encrypted parameters
(d) inserting hypercalls for converting data encrypted in one encryption scheme to another.
(e) replacing the calls to C libraries (like string operations and file operations) with AUTOCRYPT's library calls. We analyse the implementation of standard C libraries for character handling and string manipulation using AUTOCRYPT and generate precompiled AUTOCRYPT-ed libraries. We transformed 15 C library functions for evaluation of our case studies.

AUTOCRYPT transforms the annotated and well-typed IR to emit transformed un-optimized AUTOCRYPT-ed code which operates on encrypted data.

5. EVALUATION

Evaluation Goals. We aim to evaluate the effectiveness of AUTOCRYPT for following main goals:

- We evaluate the developer effort required to transform existing applications to operate on encrypted data.
- We evaluate applications to work on encrypted data using combination of various partially homomorphic encryption schemes.
- We evaluate the performance of AUTOCRYPT-ed programs compared to the traditional 'download-and-compute' strategy and use of existing fully-homomorphic encryption schemes.

5.1 Expressiveness

We select 30 case studies from common programs available on the LAMP Stack which include COREUTILS, `file` utility and three custom programs.

CoreUtils. The use of command line utilities is popular among web services like SVN [60], GitHub [33], Google shell (goosh) [34], etc. Hence, we perform evaluation on Unix file processing applications from the COREUTILS v.8.13 package [2]. Table 3 shows the programs in 8 of total 11 categories of COREUTILS that we transform to assert the effectiveness of our solution. The remaining 3

Programs	Encrypted Variables			Encrypted Constants			Changed Instructions			Compilation Time(sec)			Execution Time(sec)			Comp.to
	Enc	Tot	%	Enc	Tot	%	Enc	Tot	%	Infr.	Trans.	Total	1KB	10KB	100KB	to DC
cat(simple)	0	324	0.00	0	30	0.00	0	446	0.00	0.04	0.02	**0.06**	0.006	0.008	0.371	**0.304**
split	116	2002	5.79	5	142	3.52	1	2712	0.04	0.20	0.08	**0.28**	0.007	0.010	0.028	**0.022**
csplit	26	598	4.34	1	32	3.12	29	1217	2.30	0.27	0.32	**0.59**	0.007	0.010	0.018	**0.014**
paste	34	583	5.83	4	70	5.71	10	809	1.24	0.13	0.05	**0.18**	0.008	0.015	0.241	**0.197**
tac	78	723	10.79	1	45	2.22	1	925	0.11	0.14	0.03	**0.17**	0.007	0.010	0.280	**0.229**
expand	14	494	2.83	7	55	12.73	7	674	1.04	0.10	0.03	**0.13**	0.010	0.021	0.254	**0.208**
unexpand	25	608	4.11	7	62	11.29	7	850	0.82	0.13	0.04	**0.17**	0.008	0.034	0.348	**0.285**
truncate	0	487	0.00	0	53	0.00	0	676	0.00	0.07	0.02	**0.09**	0.011	0.021	0.062	**0.050**
shred	0	1325	0.00	0	132	0.00	0	1767	0.00	0	0.04	**0.04**	0.125	0.144	0.172	**0.141**
shuf	53	571	9.28	0	32	0.00	1	750	0.13	0.15	0.03	**0.18**	0.007	0.020	0.351	**0.288**
nl	19	529	3.59	2	45	4.44	0	648	0.00	0.08	0.03	**0.11**	0.007	0.026	0.519	**0.426**
tail	76	931	8.16	3	39	7.69	10	819	1.22	0.11	0.14	**0.25**	0.007	0.017	0.035	**0.167**
head	155	1346	11.52	4	105	3.81	3	1848	0.16	0.23	0.11	**0.34**	0.008	0.024	0.033	**0.270**
cut	58	968	5.99	6	76	7.89	20	1385	1.44	0.11	0.150	**0.26**	0.012	0.026	0.190	**0.155**
join	54	834	6.47	1	10	0.1	6	737	0.81	0.16	0.08	**0.24**	0.026	0.046	0.237	**0.194**
tr	20	721	2.77	3	53	5.63	4	592	0.67	0.06	0.06	**0.12**	0.012	0.031	0.397	**0.325**
ptx	129	872	14.79	6	62	9.62	27	1527	1.76	0.13	0.11	**0.24**	0.014	0.057	0.291	**0.239**
pr	91	1011	9.00	16	110	14.53	12	936	1.28	0.18	0.23	**0.41**	0.030	0.446	1.37	**1.124**
fmt	124	1117	11.10	20	133	15.04	30	1546	1.94	0.58	0.14	**0.72**	0.029	0.217	1.40	**1.149**
wc	61	1002	6.09	6	67	8.96	6	1381	0.43	0.25	0.06	**0.31**	0.034	0.357	2.84	**2.33**
fold	51	370	13.78	8	34	23.53	7	487	1.44	0.13	0.03	**0.16**	0.022	0.381	2.02	**1.658**
sum	36	321	11.21	0	38	0.00	13	402	3.23	0.09	0.03	**0.12**	0.020	0.036	.549	**0.450**
base64*	81	473	17.12	0	44	0.00	11	617	1.78	0.24	0.30	**0.54**	0.200	1.534	5.33	**4.376**
cksum*	34	576	5.90	0	78	0.00	24	872	2.75	0.10	0.24	**0.34**	0.371	1.632	7.78	**6.387**
file	229	2516	9.10	7	54	12.96	23	2916	0.07	0.42	0.20	**0.62**	0.186	0.120	0.194	**0.160**

Table 2: Summary of sensitivity analysis and transform pass for 25 file processing utilities. `cat(simple)` is executed without any command line options. Only `sum, base64, cksum` (shown bold faced) make hypercalls during execution. We annotate `base64, cksum` (marked star) such that multiple calls for conversion between encryption types are combined into one hypercall. Last column reports the execution time ratio of AUTOCRYPT-ed programs to 'download-and-compute' method.

categories include the programs which do not operate on the sensitive contents of file and so we do not evaluate them. Our current implementation does not support command line options involving regular expressions to programs such as `ptx, nl, tr`. However, we point out that using static memory lookup hypercall or DFA evaluation techniques, regular expression matching is possible but will require advanced analysis techniques [62]. To test the programs, we create encrypted input files of size 1, 10, 100 KB under search, Paillier and Elgamal encryption schemes. In Table 2, we present the transformation and execution time for these utilities. AUTOCRYPT transforms 25 out of 30 commands of various categories to operate on pre-encrypted files of various sizes. We confirm that the transformed programs remain functional and preserve the semantics of the original application.

File Utility. We select the UNIX `file` utility, which determines the type of a given input file, as our benchmark. We select this utility because of its usage in a web server for learning the MIME type of files and determining the HTTP content header [13]. Every file type contains a magic number at a fixed position which distinguishes it from other files. A magic file is a collection of all such magic numbers and their position for various file types. We encrypt these magic numbers with search encryption scheme. The transformed `file` command uses the encrypted magic file to determine the file types of the sensitive files. The size of the input file does not effect the performance of this application as only the bytes at a fixed position are searched to identify the file type.

Custom programs. We select three particular applications to show the broad applicability of AUTOCRYPT for functions which are used for various privacy preserving computations [36,47]. Following are the programs which we include as our case studies.

(a) Matrix multiplication : Matrix multiplication is commonly used in AES encryption. The inputs to this program are encrypted with Elgamal encryption scheme. The number of hypercalls is proportional to the matrix size. We evaluate on a 5×5 matrix which makes 25 hypercalls.

```
# where plaintext at p[i] and q[i]
#can be either 1 or 0
pai8:p[n];
int :q[n];
pai8:dist;
for(i = 0; i < n; i++) {
  tmp1 = xor_pal1(p[i],q[i]);
 dist=add_pal8(dist,tmp1);
 }
```
(i)

```
pai8:p[n],q[n];
pai8:dist;
 for(i = 0; i < n;i++)  {
   tmp1 = sub_pal8(p[i],q[i]);
   tmp2:egml =convert(tmp1:pai8);
   tmp3 = mul_egml(tmp2,tmp2);
   tmp4:pai8 = convert(tmp3:egml);
   dist =add_pal8(dist,tmp4);
  }
```
(ii)

```
egml:first[m][p],sec[p][q];
pai8:ans[m][q];
 for ( c = 0 ; c < m ; c++ )   {
    for ( d = 0 ; d < q ; d++ ) {
      for ( k = 0 ; k < p ; k++) {
       tmp1 = mul_egml(first[c][k],sec[k][d]);
       tmp2:pai8 = convert(tmp1:egml);
       ans[c][d] = add_pal8(ans[c][d],tmp2) ;
     } } }
```
(iii)

Figure 7: Pseudocode of AUTOCRYPT-ed custom programs (i) Hamming Distance (ii) Euclidean Distance (iii) Matrix Multiplication

(b) Square of Euclidean distance: We evaluate a program for computation of square of Euclidean distance which is used in face recognition applications [47]. The hypercalls are directly proportional to the number of points used for calculating the Euclidean distance. We calculate the distance for 10 points that require a total of 20 hypercalls.

(c) Hamming distance : We calculate the hamming distance between an encrypted and a plaintext input. No hypercalls are required for this computation.

We transform these program using AUTOCRYPT and report the execution time in Table 4. The transformed pseudocode for these programs are shown in Figure 7.

Custom Program	Encrypted variable (%)	Changed Instruction (%)	Time (sec)		Exec (sec)
			Infr.	Trans.	
Matrix Multiplication	12/129 (**9.30**)	75/154 (**48.70**)	0.27	0.05	0.088
Euclidean distance	15/98 (**15.3**)	50/138 (**36.23**)	0.22	0.11	0.074
Hamming distance	17/108 (**15.7**)	20/124 (**16.12**)	0.17	0.04	0.042

Table 4: Performance of custom programs. Last column reports the execution time in seconds.

Handling FHE cases. Out of the 25 COREUTILS and 3 custom programs which we transform using AUTOCRYPT, 5 applications make hypercalls to switch between different homomorphic encryption schemes. If not for our design, these applications require the capabilities of fully homomorphic encryption scheme to operate on encrypted data. Matrix Multiplication, Euclidean Distance, `sum`, `cksum` and `base64` bold faced in Table 2 and Table 4 respectively, contain operations which require switching between encryption schemes. Using our trusted hypervisor architecture and the hypervisor APIs, AUTOCRYPT can transform these applications to execute with a performance acceptable for practical usage. The execution time for applications increases linearly with the number of hypercalls. The number of hypercalls can be reduced if conversion between encryption schemes is performed in groups instead of per character. Hence, we annotate `cksum` and `base64` such that multiple hypercalls for conversion between encryption types are combined into one hypercall. However, the performance is still better as compared to requesting the client for conversion between encryption schemes. A single hypercall takes around 2 ms in our implementation —which is 3 times faster as compared to sending bytes to the client.

Untransformed applications. We discuss 5 applications from the COREUTILS which either do not type-check according to our system or require homomorphic operations that leaks a lot of information about the sensitive data. Hence, these are beyond our security design.

- The `cat` command (when executed with 'show-nonprinting' option) and `od` command requires a conversion from Paillier to search encryption. According to our type-system, such conversion leaks information (see Section 3), thus is unsafe. Note that, AUTOCRYPT can still transform these commands, however allowing such transformations will reduce the security of our system.

- The `sort` command requires the input file encrypted under order preserving scheme (OPE) as it performs relational operations (<, >) on the sensitive inputs. OPE is a weak encryption schemes as it reveals the order of encrypted data [50]. Combining OPE with search encryption scheme leaks considerable amount of information which lowers the security guarantees of AUTOCRYPT.

- `comm` and `uniq` does an equality check operation on two characters, both from the input file. To support this functionality we require the input files encrypted under deterministic encryption scheme. However, this scheme is not semantically secure and keeping a copy of this file downgrades security.

We point that for a setting where compromising semantic security of sensitive files is tolerable, AUTOCRYPT can transform the above applications to run on sensitive inputs encrypted in expected partially homomorphic encryption scheme.

5.2 Reduction in Developer Effort

We claim that AUTOCRYPT reduces the amount of developer effort required for transforming applications to operate on encrypted data. We demonstrate the reduction in developer effort by reporting the following evaluation

- fraction of the encrypted constants and variables
- fraction of the LOC modified
- the compilation time of the transformed applications using AUTOCRYPT

Table 2 summaries the analysis and execution results for 25 Unix file processing applications. For all our case studies, the number of SSA variables range from **300 to 3000**, and only **9-10%** of them are eventually encrypted. From the pool of instructions generated in the LLVM IR only **1%** of the them are changed during the transform pass. The number of constants encrypted is around **7-8%** of the total constants. These figures show that only a small amount of variables, constants and instructions are modified during the transformation. However, manually analysing a particular application and identifying exactly which sensitive variables need encryption is a non-trivial task for developers. AUTOCRYPT relieves the developer from this task and infers all the above details within reasonable time limit. Compilation measurements comprise of time spent for LLVM analysis, inference and transformation passes to AUTOCRYPT given application. The average time for transformation to AUTOCRYPT-ed code ranges from **40 ms to 720 ms**.

5.3 Runtime Performance

We perform the evaluation on Ubuntu 12.04 as guest VM with Xen hypervisor on a Dell Latitude E6430s machine having 8 GB RAM with Intel(R) Core(TM) i7-3520M CPU at 2.90GHz 4-core processor having cache size of 4096 KB. Table 2 reports the execution time of AUTOCRYPT-ed programs for input files of 1, 10, 100 KB sizes. Out of 25 applications, 19 execute within 1 second for all the input files. Remaining 6 applications take more than 1 second. The execution time for each command is directly proportional to the input file size. However, this also applies to programs that are executed in the unencrypted domain. Our results show that operating over encrypted data using AUTOCRYPT transformed code incurs some runtime overhead over native execution which is a few milliseconds. However, native untransformed applications do not provide any security of input data. Therefore, we compare performance of AUTOCRYPT-ed programs to already known techniques of 'download-and-compute' and fully homomorphic encryption scheme.

Comparison to Download-and-compute. We compare whether AUTOCRYPT-ed programs are faster to execute than the 'download-and-compute' mechanism. We measure the download and upload time for GitHub and DropBox file systems, for files of sizes 1, 10 and 100 KB (averaged over 10 runs of each size). The observed network latency is 4.88 ms/KB and 6.0 ms/KB for Dropbox and GitHub servers respectively as measured from our campus wireless network having bandwidth of 54 Mbps. For OpenSSL AES scheme, the encryption-decryption cycle takes 8 ms and 10 ms respectively for 100 KB file on a machine of 8 GB RAM and i7 core processor. Considering this as a baseline for the 'download-and-compute' mechanism, we compare the performance of AUTOCRYPT-ed programs. The last column in Table 2 shows the ratio of execution time of AUTOCRYPT-ed program on 100 KB file and execution time of 'download-and-compute' mechanism. Out of 25 applications, 19 execute faster and only 6 applications slow down by factor of 2 to 6 as compared to 'download-and-compute' mechanism. Overall, AUTOCRYPT results are promising, and it is prac-

tical to execute legacy applications on server with comparable performance to downloading encrypted files on client device for computation. We speculate that with faster cryptographic implementation and optimization, in the future we can achieve better execution performance using AUTOCRYPT. Our results serve as a baseline for comparison of future work in this direction.

Comparison to FHE. Publically available implementations of FHE take several minutes to compute simple integer operations, whereas AUTOCRYPT completes execution over complete file in this time [5]. We do not compare the time for executing these applications using FHE as no standard tool is available that runs existing C applications on FHE encrypted files. Our evaluation shows that combining partially homomorphic encryptions schemes to operate on encrypted data is a promising direction until performance of fully-homomorphic computation become reasonable for practical purposes.

6. RELATED WORK

Application of Partially Homomorphic Cryptosystems. Partially homomorphic encryption schemes are a promising direction for performing computations on encrypted data. SCiFI [47] and Sadeghi et al. [54] provide secure techniques for face recognition using homomorphic cryptosystems. Bianchi et al. [15] proposes computing discrete cosine transforms in encrypted domain and Kuribayashi et al. [41] proposes a secure fingerprinting protocol for images; both these techniques are based on the additive homomorphic property of encryption schemes. These work mainly use the homomorphic property of a particular encryption scheme to support secure evaluation of a target application. In AUTOCRYPT, we combine the homomorphic properties supported by various encryption schemes and investigate its use on a broad scale of applications. CryptDB [51] supports privacy preserving computation on encrypted database by combining various partially homomorphic encryption schemes. In our work, we automatically transform applications to use multiple homomorphic schemes, thereby automating the insight provided by Popa et al. work [55].

Secure Multi-Party Computations. A different approach for privacy preserving computation on encrypted data is to use garbled circuits, which were first proposed in the context of secure function evaluation for two-party protocols [63]. Tools such as Fairplay [43], FairplayMP [14], VIFF [22] support secure multi-party computations using garbled circuits. TASTY [36] combines garbled circuit and homomorphic encryption schemes to enable privacy preserving two-party computations. A recent work by Holzer et al. achieves secure two-party computation for ANSI C [37]. The size of the garbled circuit (which increases for large functions) dominates the performance in these multi-party computation setting. Most of the above solutions (except [37]) require the users to specify the secure function in a special-purpose high level language whereas AUTOCRYPT directly supports a simple C language.

Type System for computation on encrypted data. Fournet et al. [28] proposes a information flow type system for homomorphic encryptions. The type system of AUTOCRYPT is somewhat similar to [28], with additional support for conversion from one encryption scheme to other. We also discuss the security guarantees provided by our type-system. Mitchell et al. proposes a Haskell based language for secure cloud computing in muti-party computation and fully-homomorphic encryption platforms [46]. Fletcher et al. proposes a compiler for converting complex program to work on inputs encrypted with fully homomorphic encryption schemes [26]. In AUTOCRYPT however, we focus on partially homomorphic encryptions for secure computation on encrypted data.

Computation using Fully-Homomorphic Encryption. Another solution for privacy preserving computation on encrypted data is to use fully-homomorphic encryption. It allows securely computing arbitrary functions on encrypted data, with the first such feasible scheme proposed by Gentry [30]. The performance of such schemes is too slow for practical purposes [31]; though techniques to make it practical are of considerable interest and are being investigated [19, 56].

7. CONCLUSION

In this paper we show the effectiveness and applicability of a new tool AUTOCRYPT in enabling homomorphic computation in existing C applications. Together with a trusted hypervisor, AUTOCRYPT-ed programs offer second line of defenses against server-side data attacks.

8. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments and suggestions. We thank Zhenkai Liang, Ee-Chien Chang for their comments on an early presentation of this work and Asankhaya Sharma for his valuable suggestions about our type system. This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-495-133. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the Ministry of Education, Singapore.

9. REFERENCES

[1] Datalog Educational System. http://des.sourceforge.net.

[2] GNU CoreUtils. http://www.gnu.org/software/coreutils/.

[3] GNU Libgcrypt. www.gnu.org/software/libgcrypt.

[4] Personal Data Protection Act 2012. www.mci.gov.sg.

[5] Scarab Library. https://hcrypt.com//scarab-library.

[6] Vaultize. www.vaultize.com.

[7] Mcafee host intrusion prevention for server. www.mcafee.com, 2012.

[8] Standards for the protection of personal information of residents of the commonwealth. www.mass.gov/ocabr/docs/idtheft/201cmr1700reg.pdf, 2012.

[9] Summary of the HIPAA Privacy Rule. www.hhs.gov, 2012.

[10] What's holding back the cloud? www.intel.com, 2012.

[11] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[12] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-Confined HTML5 Applications. In *ESORICS*, 2013.

[13] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *Security and Privacy*, 2009.

[14] A. Ben-david, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *ACM CCS*, October 2008.

[15] T. Bianchi, A. Piva, and M. Barni. Encrypted Domain DCT Based on Homomorphic Cryptosystems. In *EURASIP*, 2009.

[16] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. In *EUROCRYPT*, 2004.

[17] D. Boneh, G. Segev, and B. Waters. Targeted Malleability: Homomorphic Encryption for Restricted Computations. 2012.

[18] BoxCryptor. `https://www.boxcryptor.com/`.

[19] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping. In *Electronic Colloquium on Computational Complexity*, 2011.

[20] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium 2003*.

[21] CloudFogger. `http://www.cloudfogger.com/en/`.

[22] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography(PKC)*, 2009.

[23] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 1976.

[24] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO'84*.

[25] M. Fischetti. Data Theft: Hackers Attack. `http://www.scientificamerican.com`, 2011.

[26] C. Fletcher, M. van Dijk, and S. Devadas. Compilation Techniques for Efficient Encrypted Computation. In *Cryptology ePrint Archive, Report 2012/266*.

[27] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. In *ACM SIGPLAN*, PLDI '02.

[28] C. Fournet, J. Planul, and T. Rezk. Information-flow Types for Homomorphic Encryptions. In *CCS '11*.

[29] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[30] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *41st Annual ACM Symposium on Theory of Computing*, 2009.

[31] C. Gentry and S. Halevi. A Working Implementation of Fully Homomorphic Encryption. In *EUROCRYPT*, 2010.

[32] C. Gentry and S. Halevi. Implementing gentry′s fully-homomorphic encryption scheme. In *EUROCRYPT*, 2011.

[33] GitHub. `https://github.com`.

[34] Goosh. `http://goosh.org`, 2008.

[35] B. Hardekopf and C. Lin. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*.

[36] W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM CCS*, 2010.

[37] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *CCS '12*.

[38] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Usenix Security Symposium*, 2011.

[39] T. Kim and N. Zeldovich. Making Linux Protection Mechanisms Egalitarian with UserFS. In *Usenix Security Symposium*, 2010.

[40] B. Kopf and M. Durmuth. A Provably Secure And Efficient Countermeasure Against Timing Attacks. In *IEEE Computer Security Foundations Symposium*, 2009.

[41] M. Kuribayashi and H. Tanaka. Fingerprinting Protocol for Images Based on Additive Homomorphic Property. In *IEEE Transactions on Image Processing*, 2005.

[42] LLVM. `http://llvm.org`.

[43] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *USENIX Security Symposium*, 2004.

[44] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Security and Privacy*, 2010.

[45] J. M. McCune, B. Parnoy, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[46] J. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow Control for Programming on Encrypted data. In *IEEE Computer Security Foundations Symposium(CSF)*, 2012.

[47] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - A System for Secure Face Identification. In *Security and Privacy 2010*.

[48] P. Paillier. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, 1999.

[49] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Security and Privacy*, 2009.

[50] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *IEEE Symposium on Security and Privacy*, 2013.

[51] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[52] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: Toward Data Confidentiality in Storage-intensive Sloud Applications. SOCC '11.

[53] A. Sabelfeld and A. Myers. Language-based Information-flow Security. *Selected Areas in Communications, IEEE Journal*, 2003.

[54] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient Privacy-Preserving Face Recognition. In *ICISC*, 2009.

[55] M. Shah, E. Stark, R. A. Popa, and N. Zeldovich. Language Support for Efficient Computation over Encrypted Data. In *Off the Beaten Track Workshop*, 2012.

[56] N. Smart and F. Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. Cryptology ePrint Archive, Report 2009/571, 2009.

[57] G. Smith. Principles of Secure Information Flow Analysis. In *Malware Detection*. 2007.

[58] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data . In *21st IEEE Symposium on Security and Privacy, Oakland, CA*, 2000.

[59] E. Stefanov, E. Shi, and D. Song. Towards Practical Oblivious RAM. *CoRR*, 2011.

[60] SVN. `http://subversion.tigris.org`.

[61] A. Vasudevan, S. Chaki, L. Jia, J. M. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy*, 2013.

[62] L. Wei and M. K. Reiter. Third-Party Private DFA Evaluation on Encrypted Files in the Cloud. In *Computer Security ESORICS 2012*.

[63] A. C. Yao. Protocols for Secure Computations. In *23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982.

[64] Y. Yu, J. Leiwo, and B. Premkumar. A Study on the Security of Privacy Homomorphism. In *ITNG 2006*.