# Encrypted Execution

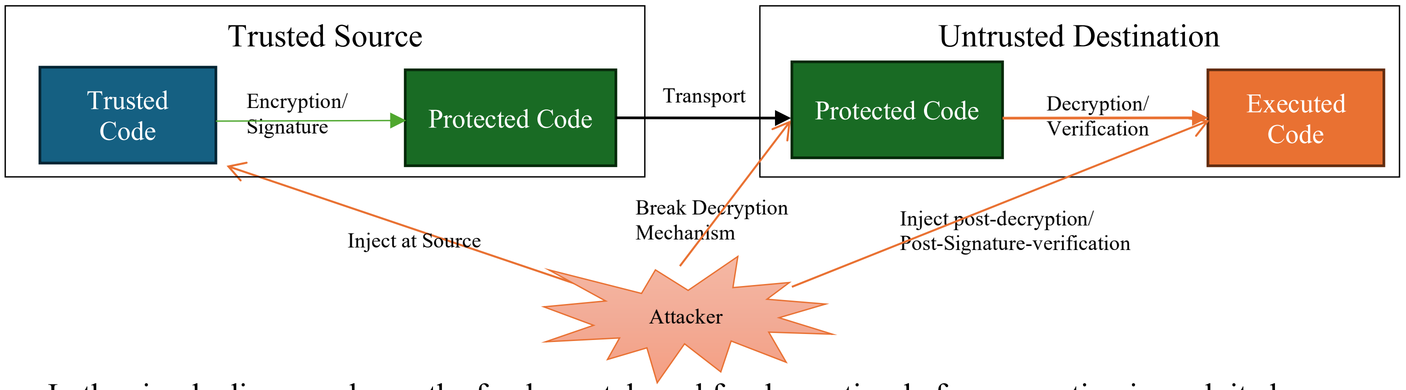
Author: Archis Gore (archis@encrypted-execution.com)
https://encrypted-execution.com

## I The need for Decryption

The fundamental problem with all Encryption mechanisms is that any encrypted data, especially executable code, must be decrypted to be useful. This paper suggests eliminating the need for decryption altogether.

Digital signatures ensure the trustworthiness of executable code but have limitations: (a) post-verification modifications are undetectable, and (b) roots-of-trust remain perpetually secure to allow future modifications.

Despite the most sophisticated Trusted Computing initiatives, Code Injection attacks consistently rank in the OWASP[i] Top 10[ii] (#1 for 15+ years until recently), with most of the top 10 vulnerabilities such as Software and Data Integrity Failures being a synonym for Code Injection.



In the simple diagram above, the fundamental need for decryption-before-execution is exploited through three main vectors:

1. **Hijack Encryption/Signing (aka supply chain attack):** The SolarWinds attack on the US Government is the most recent widely known example of this. In essence, trick the encryption/digital signature mechanism to encrypt/sign otherwise unintended code so the recipient implicitly trusts it. This takes a few different forms – an attacker may hijack a root-of-trust, steal the signing key, etc.
2. **Hijack decryption key/verification mechanism:** The counterpart to this is tricking the decryption/signature verification mechanism to trust code otherwise unauthorized code. This can be done directly by causing it to trust code, or by indirectly causing it to trust an otherwise unauthorized root-of-trust that signed/encrypted said code. BootHole is the canonical vulnerability that broke all Trusted/Secure Computing (especially on cloud servers.)
3. **Inject post decryption/verification:** Finally, the simplest and most common of all exploits – since code MUST be decrypted in order to be executable, exploit it after it

is decrypted and verified to be trusted. This is analogous to simply screen-recording a DRM-protected movie. The examples of this are numerous – PHP code injection is frequent in Wordpress, Javascript code from multiple trusted sources, frequently run in the same context. Even validated "trusted" code, can simply fetch malicious code from the internet and "eval" it.

This paper proposes Encrypted Execution, a methodology to produce infinite pairs of "encrypted execution runtime" and "encrypted code targeting said runtime," without incurring runtime performance penalties, allowing each instance to be made unique.

To illustrate some use-cases:

1. Closed-loop systems (such as Banks, Airline systems, Point-of-Sale terminals, ATMs, Embassies) could be effectively systemically isolated providing an intrinsic level of defense even if someone smuggles a USB thumb drive, or utilizes a non-physically tangible vector such as Bluetooth, Wifi, IrDA, etc.
2. Programs could be targeted for a particular instance of an OS such that they couldn't hop/jump across instances. This would curb any executables from jumping – including malware. Very effective for highly secure datacenters, or especially vulnerable field-deployed devices such as weapon systems or military assets.
3. Applied to processor Instruction Set Architectures (ISAs), even the most low-level programs such as OS kernels, bootloaders, etc. could be targeted to run on restricted machines.
4. Each browser tab (or iframe) could have an Encrypted Javascript runtime, that only executes targeted encrypted apps, that execute without decryption.

## II Encrypted Execution: An Overview

An overview of Encrypted Execution will provide context for the deep dives that follow. This overview requires that you stipulate several axioms upon which the method is built, but the deep dives will demonstrate the truthiness of said axioms.

Encrypted Execution is built on the idea that all execution runtimes are Languages, and there can be an infinite number of languages to represent the same semantics. Runtimes have chosen one specific language that's uniform across all instances.

By generating an arbitrary new language for an instance of a runtime, and translating intended code into this instance's language, the code is executable on that instance only, achieves the same semantic outcomes, at the same performance, as the unencrypted code running on the default runtime.

An intuitive way to visualize this is to imagine a parallel universe wherein the people of Earth spoke Klingon for generations, and the developers of PHP developed it to use the Klingon alphabet, with Klingon-inspired syntax, then the Klingon-ish PHP code executing on the Klingon-ish runtime wouldn't know any better. It would still be parsed with the same computational complexity (Big-Oh) and perform the same computations.

However, if a PHP developer from our universe were transported into the Klingon universe, any of their attempts to write PHP programs in this hijacked universe would fail trivially.

The rest of this paper demonstrates that:

1. A language L at any layer (microprocessor, OS APIs, calling conventions, bytecode, symbols, and interpreter) can be transformed into a never-before-seen language E generated dynamically.

   **L -> E**

2. A program LP written for Language L can be transformed into an Encrypted Program EP.

   **LP -> EP**

3. Such that EP runs on runtime E with the same algorithmic complexity that LP runs on L.

   **EP on E == LP on L**

4. That simply knowing EP and E, it is computationally difficult to modify write a new program or modify EP such that it runs on E, since isomorphic graph comparisons are already non-trivial, and we can transform L into E such that L's AST is a subgraph of E's AST, making E not a strict isomorph of L.

5. With an Apache/PHP-licensed open-source PHP implementation demonstrating this is feasible to implement, and trivial to productionize.

6. Finally, that it does not introduce any new vulnerabilities, i.e. an Encrypted Execution runtime is strictly at least as secure as an Unencrypted Execution runtime and only improves on the baseline security posture.

## III Encrypted Execution: A paradigm shift

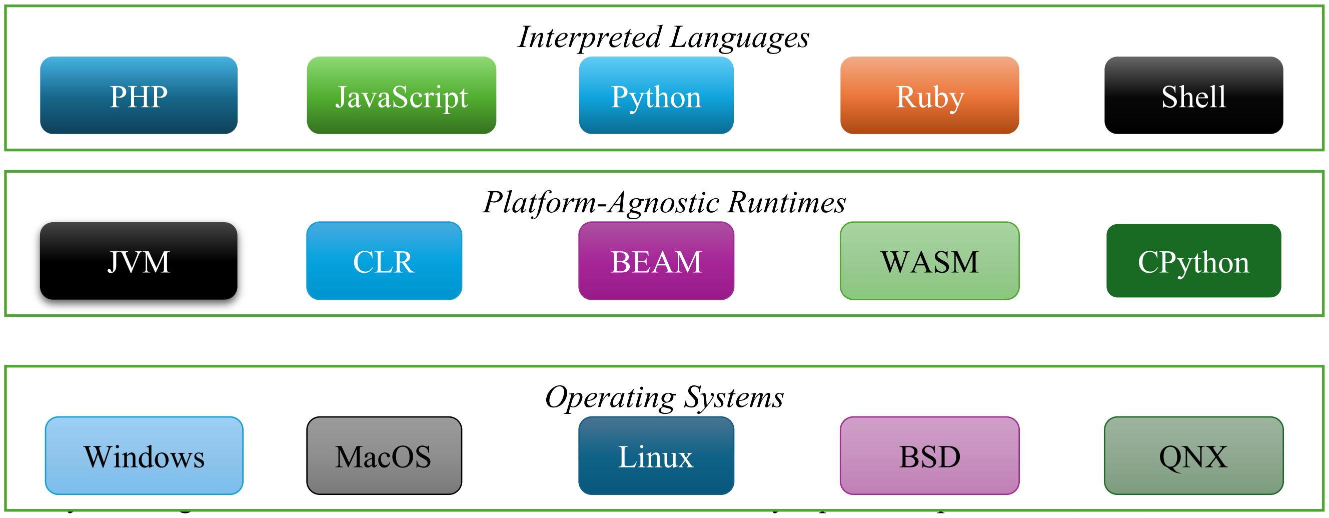
The concrete way to think about this is:

**a)** if we create the same difference that exists between an ARM, an x86, and a RISC-V processor, but we do so across various instantiations of an ARM, x86 or RISC-V processors, then ➔

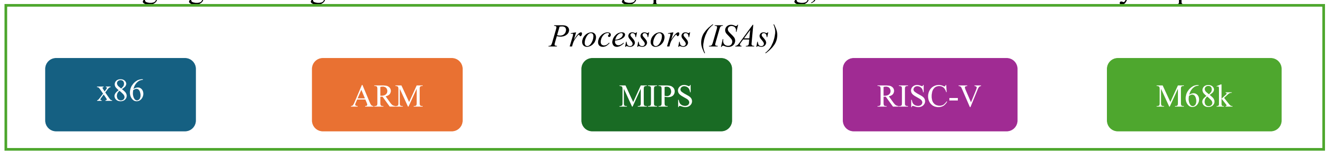
**b)** a malicious code injector faces the same difficulty as they would face coming upon the first RISC-V processor having never seen one, nor example machine code, nor an instruction set manual.

Applying this methodology at every layer of the stack, such as an Operating System's internal structures, the syscall ABI, dynamic linking ABI, static-linking ABI, and all the way up to platform-agnostic and interpreted runtimes such as the JVM, .Net, WASM, or JavaScript, we can create the same gap an attacker would have of running .Net code on the JVM, or a MacOS program on Windows, or a Python Script on Ruby.

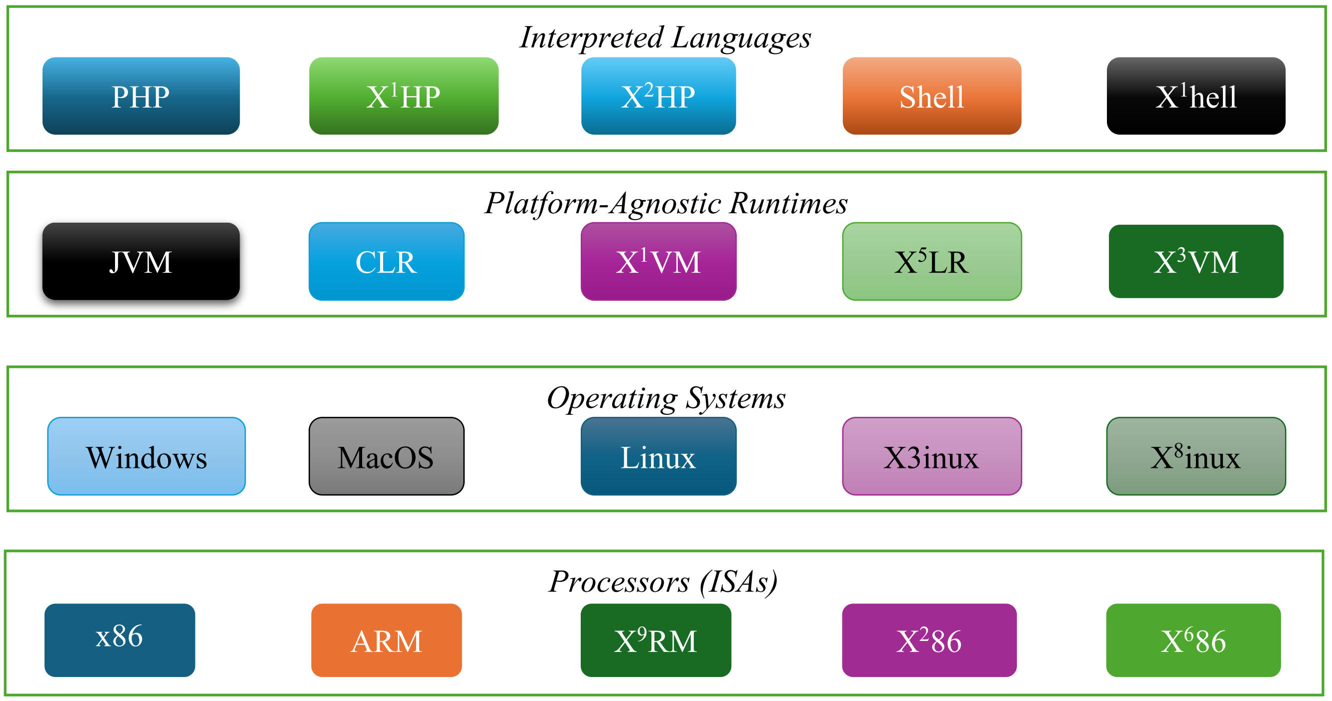
***If executing code built for one is hard to execute on another…***



and languages having the same translational gap in existing, we can make it virtually impossible



***Then we should be able to generate an infinite number of incompatible variations for each of these layers that retain the difficulty for any code not simultaneously translated to match***



All of this sounds seemingly complex, but it really doesn't have to be, if we approach it systematically. In fact, it's a lot simpler and easy to understand than you might imagine.

## IV The Encryption Process

### Everything is a Language

The process relies on two premises:

1. All executable code is a Language – all the way from machine code executed on silicon, all the way up to an interpreted language is a layering of languages.
2. Encryption is a Translation from a Usable Language into an Unusable one. Instead of Decrypting the Unusable Program, we encrypt the Runtime to make it Usable in Encrypted form. This is the programmatic equivalent of the Navajo Code Talkers.

The only reason we require Decryption at all, especially for data, is because it is prohibitively expensive for a person to learn to natively understand encrypted data. Computers are not restricted by this cost.

While producing a custom ISA for each computer may sound expensive, under narrow scenarios it may be feasible: Embassies operating in hostile territories, highly secure military devices, satellite networks, self-driving cars and so on. For highly secure applications it may even be feasible to reprogram FPGAs on-the-fly to interpret new instruction sets.

Software is orders of magnitude cheaper. For example:

1. Compiling a complete Linux (including all extended package repositories), costs as little as $20/instance.
2. Compiling a simple interpreter like PHP takes minutes on a low-performance laptop.

A PHP reference implementation demonstrating most of the techniques is provided here:
https://github.com/encrypted-execution/php

Let's start with a simple C-like interpreted language:

```
int main() {
    a = 10;

    if (a > 5) {
        a += 3
    } else {
        a -= 1
    }
}
```

There are 3 distinct features that make a language a language, and we can modify all 3 of them.

## Vocabulary

This is the most obvious way to generate a new language. If we replace the word for “book” with “goobledygook”, we’ve changed the language. This isn’t terribly difficult to reverse but it’s a start. We can replace keywords such as “int”, “for”, “if”, “else” with other words and symbols.

For example, if we substituted:

1. Int -> goobledygook
2. ( -> [
3. ) -> ]
4. if -> fndcv
5. else -> kfef
6. { -> *
7. } -> /

We get:

```
goobledygook main[] *
    a = 10;

    fndcv [a > 5] *
        a += 3
    / kfef *
        a -= 1
    /
/
```

## Syntax

Syntax is what truly makes a language a language – it determines in what order the words fit, which makes it significantly harder to reverse engineer, as it may not necessarily be a 1:1 substitution.

A trivial example is the “+=” operator in C, which can always be expanded as “a = a + 3”, or vice versa, or something entirely different with more or fewer symbols.
This is where statistical analysis and correlation begins to suffer.

For example,

1. “<x> += <n>” becomes “& <n> <x> –“
2. “if (<condition>) {“ becomes “(<condition>) fndcv *”

```
goobledygook main[] *
    a = 10;

    [a > 5] fndcv *
        &  3 a -
    / kfef *
        a -= 1
    /
/
```

This is beginning to look difficult to translate, especially if all you had was the ciphertext.

## Semantics

Semantics changes the very meaning of what is being communicated. In human language the equivalent would be sarcasm: "He inherited quite a modest sum" indicates a hefty inheritance. "I persued through the book" colloquially indicates a superficial look.

1. The obvious easy ones are double-negatives (substituting "not not <condition>" to replace "<condition>"), semantically equivalent constructs (replacing if-then statements with switch-cases and vice versa), merging or decomposing functions (i.e. f(g(x)) collapsed into e(x), and splitting h(x) into i(j(x))), and so on.
2. We can split vocabulary and syntax across multiple statements to produce contextual meaning. In essence we project the language into a higher-dimensional space (adding more keywords). This is a well-known technique used in traditional prime-factorization cryptography to good effect.
    a. 'if' is interpreted only as a conditional if there a certain keyword is present in the condition – which can be contextual.
3. We can learn from esoteric languages like INTERCAL which introduced the idea of requiring a program to say "PLEASE" a certain amount but not too much. We call these landmine keywords. At encryption-time the placement and frequency of these words is easily determined over the closure, but an attacker would likely trip up.

Let's put this all together in our code example:

1. Let's replace "if" with "switch-case", but with the syntax: "switch (value) {case 1: <do stuff>; case 2: <do stuff>;}" -> "(value) {<do stuff> when case 1; <do stuff> when case 2;} switch"
2. Replace all positive conditions with double-negatives: "<cond>" -> "not (not <cond>)"
3. Let's replace "switch-case" vocabulary with:
    a. "switch" -> "fjafg"
    b. "case" -> "hgfh"
4. Each statement must have exactly the number of "please"'s injected intermittently as the 1/2 number of tokens in that statement (rounded down) with a minimum of 1, and no two pleases may appear consecutively.

Applying semantic transforms to original C gets us:

```
int main() {
    a = 10;

    (not (a <= 5)) {
        a += 3
        break;
        case true:
        a -= 1
        break;
        case false:
    } switch
}
```

Applying all the other transforms including vocabulary and symbols, we get:

```
goobledygook please main please [] please *
    a = please 10;

    (not please (a please <= please 5))please *
        & please 3 a please -;
        please break;
        hgfh please true:
        a please -= 1;
        please break;
        hgfh please false:
    / please fjafg
/ please
```

Now we've created a code-injector's nightmare as well as a semantic analyst's nightmare. This is a manually generated example, but through code, the complexity gets significantly higher for very little one-time translation cost.

For brevity a lot of other transformation techniques are omitted from this whitepaper, but they can be seen in the open source prototype implementation available at github.com/encrypted-execution.

## Cardinality

As we all know, the exact same shuffle of a deck of 52 playing cards has never repeated before and will never repeat in the lifetime of the universe.

A programming language has hundreds, if not thousands of symbols. We have listed 5 out of dozens of transformation methods. It is intuitively obvious that the same language can never be generated twice.

In fact, given the highly parallelized transformation implementation, it is more difficult in practice to reproduce a particular scramble, given the same seed.

## V Ensuring Correctness in a Closure

A frequent misunderstanding of EE is to assume that any arbitrary program LP from source Language L, needs to always be correctly and conclusively transformed into the Encrypted Language E. Language translation is hard, and transforming arbitrary programs from L to E is arbitrarily hard.

However, just as the target use-cases of EE are limited, so are the source programs that need to be Encrypted for any one target instance. Even if the space of Source Programs is large (including an entire Linux distribution, including extended repos), it is still countably finite.

The human language analogy to this is as follows. If we were asked to generate an imaginary new language for a TV show, our made-up language at least and at most only needs to express the dialogue within the script. It does not need to express everything possibly expressible across any human language, nor in any single human language, nor all that has already been expressed in a single human language. This makes the problem far more tractable than initially seems.

Let's call our Execution Runtime E and our Plaintext code P. Let's use the Equality sign to denote that code C can execute on runtime E. We begin with the state:

P = E (i.e. P executes on E)

Let's say we apply one of the N possible transforms first – and during the transform we parse P into its AST to ensure we can resolve P's semantics correctly. We also apply the same transform to the parser of E ensuring we create no contradictions in parsing.

Therefore, we get intermediate representation:

P1 = E1 (i.e. P1 executes on E1)

We continue applying from our set of transformations, at each n'th step, ensuring that:

Cn = En (i.e. transformed code Cn executes on En, and is semantically same as the original C)

We can continue transforming until we can find no more opportunities to transform, or we have reached a sufficiently high 'n'.

The same is applicable for ABI transforms (i.e. substitute registers in calling conventions, reorder structs, etc.)

## VI Performance

The Big-Oh complexity of the parser remains identical to the original language parser E. There might be changes to constants if the AST is modified (semantic transforms mentioned above), but fundamentally a Recursive-Descent Parser will remain an RDP, as will LL, LR, shift-reduce and any of the others.

Based on the closed source code and choice of transformations applied, it is easy to control any performance tradeoff for the application in question.

## VII Security with the rise of AI/ML

There are three key vectors in which to consider how this methodology holds up, and it is important to realize the context in which the attacker is operating.

Before we get to that let me stipulate without argument that Encrypted Execution (and for that matter any and all Encryption) fails if:

1. Someone steals the encryption key itself
2. The authorized code itself is compromised or buggy (i.e. an Encrypted Execution Malware is still Malware. EE doesn't make a malicious program non-malicious.)
3. Given enough compute, time and examples, any encryption can be broken.

Normalizing an isomorphic graph (in this case the Abstract Syntax Tree) is not-yet-Polynomial. This gets more expensive if the graph isn't a strict isomorph but a semantic one. We run up against the Church-Turing thesis: Can a program look at another program and determine if it will ever halt? Let alone answer whether it will achieve the same outcomes as another arbitrary program.

Consider the context of the attacker. In the best case they have access to the encrypted runtime, an example encrypted program they can study, the semantics of the program already known to them. In this context they must craft an exploit string such that it is accepted by the runtime in the encrypted domain and accomplish the attacker's purpose. For this, they need to parse at least a subset of the existing program, which can be made arbitrarily computationally expensive, depending on transformations applied.

In practice, most attackers will not have this generous of context, getting at best a single-shot chance at code injection that must succeed or fail without much feedback.

In addition, any failed attempt at such a code injection would produce a conclusive signal to the defender due to attempted invalid input.

This makes our approach to Encrypted Execution at best significantly stronger than running plain code, and at worst, introduces no additional vulnerabilities that otherwise did not exist.

## VIII Immediate Applications and the Future

Applications of this technology are wide-ranging, and the applications are listed in order of feasibility as it is proven out more.

1. The easiest start today is all PHP-based websites can be encrypted for free (as-in-speech and beer) using the open source reference implementation available at github.com/encrypted-execution. This reduces Wordpress's common vulnerability of code injection.
2. API encryption applied at various levels (in-kernel APIs, system calls, dynamic linking calling conventions, inter-library calling conventions, etc.) can allow targeting of programs to run on authorized machines only. While this may not be

easily scalable for consumers, it would be a significantly strengthen closed systems such as:
    a. High-security data centers, or any facilities where electronics and USB thumb drives are banned.
    b. ATM machines
    c. Point-of-Sale terminals for Banks, Airlines, etc.
    d. Power grids
    e. Embassies
3. Web browsers that interpret an Encrypted JavaScript can be deployed initially in closed-systems such as locked-down corporate machines, Point-of-Sale terminals, hospital systems, banks and more. Eventually Web Browsers may be extended to dynamically exchange the AST they parse per-isolation zone such as a browser tab or an iframe.
4. Taking this to the extreme, it is conceivable to produce pairs of <compiler-codegen-backend, ISA> such that the compiler is able to target a custom ISA. This would mean the silicon itself would execute programs in the Encrypted Domain, and only the possessor of the compiler codegen backend would be able to target it. Applications for this include:
    a. Electronic assets deployed in hostile zones. Especially military assets that cannot be adapted by unintended users.
    b. Nation-State digital sovereignty: Nations may develop x86/ARM-transformed ISAs and may insist on compiling critical OS and software code on their soil, to mitigate against reliance on foreign technology companies and their governments.

## IX Patents, Copyrights and Prior Art

1. The patent (USPTO 10,733,303) covering this method is owned by me and pledged into the public domain. If you want a more explicit guarantee, contact me to get a perpetual unrestricted royalty free license for a nominal attorney fee to execute the contract.
2. Most of the code under the [github.com/encrypted-execution] repositories is licensed under the upstream repo licenses (due to modification clauses). Any code not covered by that is Copyrighted by the Polyverse Corporation, so you may not relicense it. All code was released under the Apache 2.0 license wherever otherwise unspecified, or default license was not applicable.
3. Writing different code from the ground up is not only straightforward, but will probably improve upon a lot of our early design decisions, easily working around the copyright.
4. Data Structure Randomization is well-studied, and implemented in the Linux Kernel since at least 2017: https://www.linuxfoundation.org/blog/blog/linux-kernel-developer-kees-cook. Other references:
    a. https://friends.cs.purdue.edu/pubs/ESORICS15.pdf

5. “Encrypted Execution” has been of interest for a while, although most of it focuses on the number theory-type Encryption (prime factorization, modulo arithmetic on elliptic curves, etc.)
    a. https://ieeexplore.ieee.org/document/7011332
    b. https://eprint.iacr.org/2023/641

---

[i]OWASP (Open Worldwide Application Security Project) is a nonprofit organization that aims to improve software security. It's a global community of experts who share their knowledge on how to build secure software.
[ii] The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.
[iii] Homomorphic encryption (HE) is a type of cryptography that allows users to perform calculations on encrypted data without first decrypting it. This method keeps the data confidential while still allowing it to be used for analysis.