- 1. Overview
  - 1.1. Workflow
  - 1.2. Artifact Hierarchy Explained
- 2. Environment Setup
  - 2.1. Docker
  - 2.2. Build and Start Docker Containers
  - 2.3. Build Components
- 3. Run Toolchain for Benchmarks

1. Overview

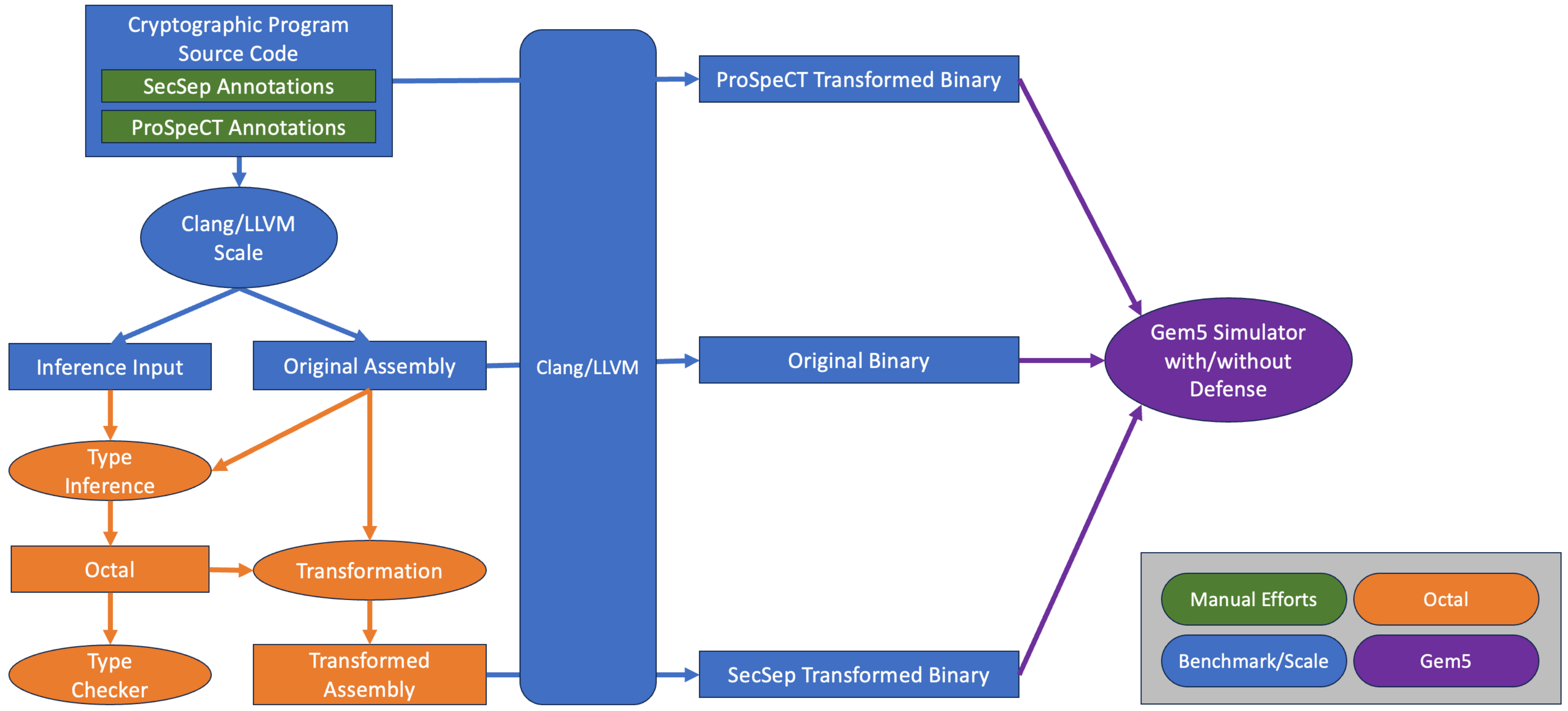
Cryptographic software, while following constant-time coding guidelines, may still leak secrets under speculative execution. For defenses where the processor tracks secrets using taint tracking and delays secret-leaking speculation, precisely tracking secrets in memory is crucial to avoid overtainting and unnecessary delays. Tracking secrets on the stack is especially challenging because secrets can mix with public data.

We present a prototype implementation of ***SecSep***, a transformation framework that rewrites assembly programs at compile time by identifying and partitioning secret and public data on the stack. After partitioning, secret and public data are divided into different pages. This enables a simple yet precise way for taint tracking to tell the secrecy of data loaded from the memory. ***SecSep*** introduces a typed assembly language *Octal* and a heuristic inference algorithm to recover lost dataflow semantics at assembly level.

Programs transformed by ***SecSep*** achieve secure speculation with an average overhead of 1.2%, on a hardware implementing simple taint tracking.

This artifact implements the entire ***SecSep*** workflow, which infers, transforms, and evaluates multiple cryptographic programs.

1.1. Workflow



SecSep Workflow

SecSep workflow can be described in five steps:

1. User writes down SecSep annotations in the source code for each function.
2. Benchmark toolchain compiles source code into assembly and generates inputs for the inference tool.
3. Type inference tool infers dependent type, valid region, and taint type, generating *Octal* as the output. Then, type checker verifies the correctness of *Octal*, and original assembly is transformed using information from *Octal*.
4. Benchmark toolchain compiles original and transformed assemblies to binaries.
5. Binaries are run on Gem5 simulator with or without the ProSpeCT-like defense to evaluate their performance.

1.2. Artifact Hierarchy Explained

The framework is composed of four components explained in the following table. We use different colors to indicate where they are in the workflow.

`$ROOT` represents the root directory where this `README` file locates.

Component	Location	Purpose
Scale	`$ROOT/scale`	A tiny AST walker based on Clang to collect source code information: (1) Help parse ***SecSep*** annotations on each function. (2) Get statistics of each function, e.g. the number of arguments.
Benchmark	`$ROOT/benchmark`	A directory to hold all benchmarks and do the compilation stuff. (1) Contain the source code of all benchmarks. (2) Use Scale to parse ***SecSep*** annotations and generate inputs for inference. (3) Compile source code into assembly; compile (transformed) assembly into binary.
Octal	`$ROOT/octal`	Inference algorithms to infer Octal, the typed assembly language, from the raw assembly code. (1) Infer the dependent type, valid region, and taint type. (2) Check the inference results using a small-TCB checker. (3) Transform original assembly based on inferred Octal using ***SecSep***'s transformations. (4) Run evaluations using (transformed) binaries and Gem5.
Gem5 Simulator	`$ROOT/gem5`	Implement the ProSpeCT-like hardware defense on O3 CPU for evaluation.

To get more information about each component, please visit `README` files under each component's directory.

2. Environment Setup

2.1. Docker

We use Docker to provide a stable environment for all components.

You need to install Docker if your OS does not have one. Installation instructions can be found at here. We recommend following the steps of 1. Click on your platform's link 2. Follow section “Uninstall old versions” 3. Follow section “Install using the apt repository”

Our test platform uses `Docker version 27.5.1`, though most versions of Docker should work. If you encounter any problem, please consider upgrade or downgrade to this version, or contact us directly for supports.

2.2. Build and Start Docker Containers

Inside `$ROOT` directory, run

```
docker compose up -d --build

# For older Docker, the command is "docker-compose" instead
```

to build the all-in-one container `secsep` for all four components. Run `docker ps` to make sure it is built and started successfully.

To attach to the container, simply run

```
docker exec -it secsep /bin/zsh
```

The entire `$ROOT` directory on the host is mapped to `/root/secsep` (or `~/secsep`) in the container. Unless otherwise specified, all following operations are performed inside the container, where `$ROOT` is assumed to be `~/secsep`.

2.3. Build Components

Among all components, Scale, Octal, and Gem5 needs to be built. However, you only need to build Gem5 manually. The other two are built automatically when they are invoked.

To build Gem5,

```
# switch to Gem5 directory
cd ~/secsep/gem5
# build Gem5 using scons
# <nproc> is at your discretion
scons build/X86_MESI_Two_Level/gem5.opt -j<nproc>
```

You will see Gem5's environment prompts to setup `pre-commit`. You can safely ignore/skip the warnings since we are only building Gem5 instead of developing it.

3. Run Toolchain for Benchmarks

Currently there are six supported benchmarks: `salsa20`, `sha512`, `chacha20`, `x25519`, `poly1305`, `ed25519_sign`. Their source code are located under the benchmark directory.

We provide commands to run ***SecSep*** toolchain on one or all benchmarks. These commands correspond to the five steps introduced in the workflow section. You must switch to the specified directory to run the commands. The results of each command is briefly explained. More information are explained in later sections.

Here are some configurable arguments and their explanation:

- `<bench>` specifies the benchmark you choose to work on. Commands without `<bench>` work on all available benchmarks.
- `<delta>` specifies the delta (in bytes) used by ***SecSep*** transformation. Must be specified in hexadecimal format. In paper's evaluation, we use `0x800000`, i.e. 8MB.

Step	Work Directory	Commands	Results
1	`~/secsep/benchmark`	***SecSep*** annotations have been written by us. You can search `"@secsep"` in the benchmark directory to get a preview of them by running `grep -rnI "@secsep"`	N/A
2	`~/secsep/benchmark`	Use `make paper -j` to compile and construct inference inputs for all benchmarks, or use `make <bench>` to work on a specific benchmark.	You can find the assembly code and compile-time information like call graph, struct layouts, and stack spill slots under `analysis/<bench>`.
3	`~/secsep/octal`	Run `./scripts/run.py full --delta <delta> --name <bench>` to infer, check, and transform one benchmark.	Infer results and logs are under `out/<bench>`. Transformed assemblies are installed to `~/secsep/benchmark/analysis/<bench>/<bench>.<transform>.s`.

Step	Work Directory	Commands	Results
		Run `--name <bench1>,<bench2>,...` to work on multiple benchmarks. To work on all benchmarks in parallel, just remove the `--name` argument.	
4	`~/secsep/benchmark`	`./scripts/get_binaries.py`	Compiled binaries are at `analysis/<bench>/build`.
5	`~/secsep/octal`	Run `./scripts/eval.py --delta <delta> -p 16 -v` to let Gem5 exeuute all original or transformed binaries to evaluate their performance. The parallelism is controlled by `-p` (FYI, there are around 42 evaluation tasks. Be careful that too much parallelism may drain your memory). Use `-v` or `-vv` to get verbose output. You will see the prompt “Will run gem5, confirm?”. Press `y` and `<Return>` to confirm running Gem5 and get brand-new evaluation results.	An evaluation directory named by current time will be generated under `eval`.
6	`/root/octal`	Run `./scripts/figure.py <eval dir>` to draw figures according to the evaluation, where `<eval dir>` is the directory generated in step 5.	Figures are under `<eval dir>/figures`.