O2 Platform's MethodStreams (2010 Open Source SAST engine)

by Dinis Cruz and ChatGPT Pro Deep Research (in 11 Feb 2025)

1. Introduction

The OWASP O2 Platform is an open-source toolkit created by Dinis Cruz as a “new paradigm” for performing and documenting web application security reviews (Dinis Cruz Blog: OWASP O2 Platform).

Unlike conventional static analysis suites, which often generate overwhelming reports, O2 was designed to **augment the security review process** by automating expert knowledge and enabling interactive code analysis (Dinis Cruz Blog: OWASP O2 Platform).

In traditional Static Application Security Testing (SAST), tools struggled with large codebases and complex data flows – they focused on scanning millions of lines of code, but often failed to *connect the traces* between source and sink across disparate modules (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size') (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size').

This meant that critical vulnerability paths could be missed if the data flow crossed file or component boundaries (common in modern architectures with web services, message queues, config-driven links, etc.) (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size').

The **problem** was not simply scan speed or volume, but the inability to trace user input to its ultimate sensitive endpoint through numerous intermediate calls and layers.

Cruz recognized that to improve security findings, the analysis needed to center on **traceability** rather than raw scan size (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size').

The motivation behind *Method Streams* and *Code Streams* was to bridge these gaps by reshaping how code is analyzed. Instead of treating each source file in isolation, O2 would extract and consolidate the relevant pieces of code that form a complete **source-to-sink flow**, making it far easier to follow the propagation of potentially malicious input.

In essence, Method Streams and Code Streams were introduced as a direct answer to the limitations of traditional SAST, enabling security testers to “connect the traces” across large applications (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size').

This document provides an overview of O2’s approach and sets the stage for deeper discussion of Method Streams and Code Streams as innovative techniques to enhance static analysis.

2. Evolution of the Concept

From Call Trees to Method Streams: In early static analysis, a *call tree* or call graph for a given function shows which other functions it invokes. Cruz took this concept a step further by materializing the call tree into actual code. A *MethodStream* is essentially **a code representation of an entire call-tree**: one file that contains the original method and *all* the methods it calls, recursively (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). In other words, rather than just visualize the calls, O2’s engine extracts the source of every method in the chain and concatenates them. Cruz first implemented MethodStreams around 2010 while reviewing a large .NET application – he faced a situation where a web service entry point was spread across “hundreds of files” of supporting code (Read Generation Z Developers | Leanpub). By programmatically gathering all those methods into a single stream, he could read *one* file and see the complete execution flow. This made security analysis far more efficient, as *only the relevant code* was in view (Read Generation Z Developers | Leanpub). According to an OWASP AppSec presentation, *“The concept of MethodStreams makes it radically simpler to get all of the source for a single method in one place to easily ‘follow the taint’.”* (Owasp o2 platform november 2010 | PPT). In practice, a MethodStream flattens a complex call hierarchy into a linear source listing – for example, O2 can take a web service like **HacmeBank’s** `CreateUser` API and produce a file containing `CreateUser` and all the functions it calls (e.g. the `DataFactory` and `SqlEngine` methods it uses) (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). With all code in one place, a security reviewer can trace an input from its entry point through each processing function to its outcome (such as a database query), without jumping between files. This ability to **“follow the taint”** end-to-end was a major evolution in static analysis workflow (Owasp o2 platform november 2010 | PPT).

Introduction of Code Streams for Taint Paths: While MethodStreams aggregate all code for a given entry point, Cruz later evolved the idea to focus on individual data flow paths within that merged code. These became known as *CodeStreams*. A CodeStream is essentially **a specific “taint” path** through a MethodStream (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). Instead of one big file of all calls, a CodeStream pulls out one sequence of method calls and variable assignments that lead from a source to a sink. In Cruz’s words, CodeStreams provide “a stream of all variables that are touched by a particular source variable” – essentially performing a static *taint flow analysis* on the MethodStream (Read Generation Z

Developers | Leanpub). If MethodStreams give the full map, CodeStreams trace the highlighted route on that map. For example, in the HacmeBank scenario, once the MethodStream for the `CreateUser` web service was generated, O2 could derive specific CodeStreams such as *"user input → validation routine → SQL query"*. Each CodeStream would be a slice of code (or an annotated sequence) showing how an untrusted input flows through sanitization (or lack thereof) to a sink. Cruz demonstrated this by identifying an **SQL Injection** path in HacmeBank: starting from an input parameter in the web service, through the data layer, into a dynamically built SQL command (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). This was captured as a CodeStream, isolating the vulnerability trace for easy inspection. In summary, MethodStreams gave a comprehensive view of all possible flows from a function, and CodeStreams distilled each *possible data flow* (good or bad) within that view (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). This evolution—from call trees, to consolidated MethodStreams, to granular CodeStreams—greatly enhanced the ability of a static analysis engine (and the human using it) to uncover how data moves through an application and where it might be exploited.

3. Technical Implementation

Generating Method Streams with ASTs (C#)

Under the hood, MethodStreams are enabled by AST (Abstract Syntax Tree) manipulation and code refactoring techniques. The O2 Platform included a powerful C# code parser and REPL environment, which Cruz repurposed to build the MethodStream generator (Read Generation Z Developers | Leanpub). The approach can be summarized as an algorithm that *traverses the call graph and extracts code*:

1. **Identify the entry method** – For example, a public web service method `X` in a .NET application (Read Generation Z Developers | Leanpub). This is the starting point (source) for the analysis.
2. **Resolve direct calls** – Find all methods that `X` calls directly (its first-level callees) (Read Generation Z Developers | Leanpub). This requires parsing `X`'s body to identify method invocation expressions.
3. **Recursive call expansion** – For each of those callee methods, determine what methods *they* call, and so on, exploring the full call tree depth-first or breadth-first (Read Generation Z Developers | Leanpub). The process continues until reaching leaf methods (methods that call no further methods of interest).
4. **Collect AST nodes** – As methods are discovered, their parsed AST representations (syntax trees) are gathered in memory (Read Generation Z Developers | Leanpub). Because O2's engine works on the AST level, it can easily clone or move syntax subtrees.
5. **Assemble the MethodStream source** – A new source file is created by pretty-printing all the collected AST nodes (methods) into code form (Read Generation Z Developers | Leanpub). In effect, this file is a *surgical copy* of code from many files, stitched

together.

Cruz described this process as *"starting on Web Service method X, calculating all methods called from X (recursively), capturing the AST of all those methods, and creating a new file with all that code"* (Read Generation Z Developers | Leanpub). The resulting MethodStream file contains **only the code needed** to understand that entry point's behavior (Read Generation Z Developers | Leanpub). Notably, this AST-driven refactoring preserves correct syntax and structure – it's equivalent to performing an automated copy-paste of function bodies, but done in a semantically correct way (leveraging the compiler's understanding of the code). The benefit is dramatic: Cruz reported that a MethodStream for one web service ended up ~3,000 lines of code all in one file, versus the 50,000+ lines spread across 20+ source files that a reviewer would otherwise have to open (Read Generation Z Developers | Leanpub). By relying on ASTs, the engine can also tweak the output. For instance, Cruz was able to insert additional relevant code that wasn't literally in the call tree – such as adding known input validation regexes at the top of the file, or appending the SQL stored procedure definitions used at the bottom – to make the review even more self-contained (Read Generation Z Developers | Leanpub). This showcases the **refactoring capabilities**: because the code is manipulated as a structured tree, O2 could merge and augment content as needed (something impossible with a simple text grep).

To illustrate in C# pseudo-code, the MethodStream creation might work like this:

```
// Pseudo-code for MethodStream generation
MethodDefinition entryMethod = AstParser.ParseMethod("Service.cs", "CreateUser");
var methodsToInclude = new HashSet<MethodDefinition>();
var worklist = new Queue<MethodDefinition>();
worklist.Enqueue(entryMethod);

while (worklist.Count > 0) {
    var method = worklist.Dequeue();
    if (!methodsToInclude.Contains(method)) {
        methodsToInclude.Add(method);
        foreach(var callee in method.CalledMethods) {
            // Resolve callee definitions via AST or reflection
            MethodDefinition calleeDef = AstParser.GetMethodDefinition(callee);
            if (calleeDef != null)
                worklist.Enqueue(calleeDef);
        }
    }
```

```
}
// Now pretty-print all collected method ASTs into one file:
string methodStreamCode = AstPrinter.Print(methodsToInclude);
File.WriteAllText("CreateUser_MethodStream.cs", methodStreamCode);
```

In O2's actual implementation, the AST parser (backed by a C# compiler or Roslyn/NRefactory) handles the heavy lifting of `ParseMethod` and printing. The key point is that by using the AST, the integrity of the code is maintained. As Cruz notes, achieving a *round-trip* (parse → modify → regenerate) with no changes to the original semantics is crucial in refactoring (Read Generation Z Developers | Leanpub). This ensured that the generated MethodStream file could be compiled or analyzed just like any other source file, and that reading it felt natural to a developer. The concept of MethodStreams is thus an application of **automated code refactoring**: it gathers slices of multiple files and merges them into one – an operation made possible by treating code as data (the AST) rather than text (Read Generation Z Developers | Leanpub).

Example: Using a MethodStream in .NET

Consider the earlier example of the `CreateUser` web method in the HacmeBank application. Using O2, a security tester would generate a MethodStream for `CreateUser`. O2's engine would parse `CreateUser` (from the WebService class) and find that it calls a `DataFactory.CreateAccount` method, which in turn calls `SqlEngine.ExecuteNonQuery` to run a database insert (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). All three of these method bodies (plus any others along the way) would be pulled into a single C# file, say `CreateUser_MethodStream.cs`. The tester can open this file in an editor and see something like: first the code of `CreateUser`, followed immediately by the code of `DataFactory.CreateAccount`, followed by `SqlEngine.ExecuteNonQuery`, etc., as if they were all originally written in sequence. Any library calls that couldn't be resolved to source (e.g. external .dll methods) might be stubbed out as auto-generated placeholders (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method), but the core business logic is all there. Now, with this MethodStream in hand, the reviewer can manually follow the data flow: e.g. an input parameter `username` in `CreateUser` is passed into `DataFactory.CreateAccount`, which eventually concatenates it into an SQL string in `SqlEngine.ExecuteNonQuery`. The vulnerability (an SQL Injection) becomes apparent by reading straight through, whereas it might have been overlooked if one had to open each file separately and mentally link the flow. In O2's interface, one could also load this MethodStream file back into the analysis engine – essentially treating it as a mini-project to run static analysis on. By scanning the consolidated file, any data flow from `username` to a database call would show up as a single connected trace, something that many SAST tools would miss when the code is split across components. Cruz notes that his approach was to create these files and **"then only scan them"**, greatly simplifying large-scale analysis (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). This way, the scanning

engine deals with a focused chunk of code (the trace of interest) rather than the entire codebase, reducing noise and improving trace completeness.

Adapting Method Streams for Python

While the original MethodStreams were implemented for C# in the O2 Platform, the underlying concept can be applied to other languages. The prerequisite is the ability to parse code into an AST and navigate function call relationships – capabilities that many modern languages have. **Python**, for instance, has a built-in `ast` module that can parse source code into an AST. A Python implementation of MethodStreams would involve iterating over the function call graph in a similar way: starting from a chosen entry function, gather all the functions it calls (which might reside in different modules), and then output a merged Python script containing those function definitions. Dinis Cruz's work emphasizes seeing "code as a graph" of objects that can be manipulated (Read Generation Z Developers | Leanpub), and this is language-agnostic. Using Python's AST, one could load the target module, find the AST node for the function of interest, then recursively resolve any `ast.Call` nodes within it to their function definitions (perhaps by importing the other modules and parsing them as needed). The end result would be a new Python file that, for example, takes a Flask route handler and inlines all the utility functions it uses across the project. This would allow a security analyst to inspect, say, how user input flows through a series of helper functions in one consolidated view, just as MethodStreams do for C#. The idea of merging "slices of multiple source code files" is equally powerful in a dynamic language like Python (Read Generation Z Developers | Leanpub). There are extra challenges (e.g. Python's dynamic typing and introspection might make call resolution harder), but the principle stands. In summary, the MethodStream technique can be re-implemented in Python by leveraging its AST and module import system to achieve the same source-to-sink visibility that O2 provided for .NET code. This demonstrates the broad applicability of Cruz's approach – by combining parsing and refactoring, we can enhance static analysis in any programming ecosystem.

4. Applications in Security Testing

Improved Vulnerability Traceability

Method Streams directly tackle one of the hardest aspects of static analysis: tracking data flows through layers of an application. By assembling all relevant code into one place, MethodStreams make it much easier to trace vulnerabilities such as SQL injection, cross-site scripting, or unsafe file access. In a traditional SAST report, a single vulnerability might be represented by a chain of steps scattered across different files and separated by "gaps" (where the tool lost the trace). O2's MethodStreams close those gaps by design – the entire chain is laid out sequentially in the MethodStream file, so nothing gets lost. As Cruz explains, large applications often have "tons of air-gaps" due to interfaces, web services, queues, config files, reflection, etc., which cause conventional scans to

miss connections (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). MethodStreams mitigate this by *statically inlining* those connections (when possible) so that, for example, a value put into a message queue in one component and read out in another can be viewed together. This significantly improves vulnerability detection accuracy. The approach shifts the problem from “Can the tool follow the flow through all these hops?” to “We’ve already connected the hops, now let’s examine the flow.” Indeed, Cruz notes that many commercial SAST engines struggled with trace continuity and would resort to dropping complex traces or simplifying analysis for large codebases (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'), resulting in missed issues. By contrast, using MethodStreams, O2’s analysis was able to handle large sets of findings more gracefully, since each MethodStream file was a coherent, smaller unit focused on one entry point (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). In practice, security reviewers found that using MethodStreams led to more complete findings – if a source input reaches a sensitive sink anywhere in the application, the MethodStream containing that path will make it evident. The ability to “connect the traces” after modular scanning is what sets this approach apart (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). It bridges what static analysis traditionally left for the human to figure out. Moreover, it aids manual code review: reading a MethodStream is like reading a detailed story of an execution path, which is far quicker and less error-prone than opening file after file in an IDE and trying to follow variable values mentally. Cruz recounted that some of his generated files, though large (~3k lines), were *massive simplifications* of the code navigation needed – they saved him from combing through perhaps 20 separate files and 50k lines of code (Read Generation Z Developers | Leanpub). This focused view allows testers to spot missed validation, improper error handling, or dangerous calls along the path much more readily. In summary, MethodStreams improve vulnerability traceability by ensuring that **source-to-sink paths are preserved and visible**, thereby increasing the chances of detection for complex, multi-step vulnerabilities.

Case Study: HacmeBank Analysis

One of the prominent demonstrations of MethodStreams and CodeStreams was with OWASP’s HacmeBank, an intentionally vulnerable banking application. Cruz used O2’s SAST engine on HacmeBank to show how an attack scenario can be uncovered step by step. In the HacmeBank web service `CreateUser` example, the MethodStream gathered all the code involved in creating a new user – from the web service endpoint, through the data factory, down to the SQL engine (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). Within this consolidated code, O2 then identified an input that flowed into a database query without proper sanitization. By selecting that input as the starting point, the tool generated a CodeStream highlighting the **exact sequence of calls and operations** that led to an SQL Injection vulnerability (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). Cruz’s blog illustrates this with a series of screenshots, each stepping through the taint propagation: (1) the user-provided input enters the system, (2) it passes through certain business logic, (3) it reaches the database query builder, and so on, until (N) the unescaped input is concatenated into an SQL statement (Dinis Cruz

Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method) (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method). This step-by-step trace (essentially the CodeStream) made it immediately clear how the vulnerability could be exploited. Such clarity is often lacking in raw static analysis outputs, where one might just get a list of line numbers or a call stack that is hard to follow. In the HacmeBank case, the MethodStream/CodeStream approach not only found the vulnerability but also produced an *explanatory narrative* of it. This case study underscored the value of O2's approach: even in an application with many layers, the critical data paths could be unraveled and examined in a straightforward manner. The HacmeBank demo was turned into a full proof-of-concept (PoC) where the findings from the static analysis were quickly validated by crafting an exploit – something O2 also facilitated by integrating with testing tools (Owasp o2 platform november 2010 | PPT). As David Campbell noted in an OWASP review of O2, having the MethodStream to follow the taint made it much easier to jump from static analysis to actually exploiting the issue (Owasp o2 platform november 2010 | PPT) (Owasp o2 platform november 2010 | PPT). This tight feedback loop (find in code → confirm in running app) is a powerful practice in security testing. Beyond HacmeBank, Cruz and others experimented with MethodStreams on real-world enterprise apps and found that it scaled well: for each entry point (e.g., each API or each UI event handler), a separate MethodStream could be generated and analyzed. This modular approach meant that very large applications could be broken down into dozens or hundreds of more digestible pieces, each piece giving insight into one functionality's security posture.

Scaling Static Analysis with Modular Streams

MethodStreams offer a pathway to **large-scale static analysis** by analyzing big systems in smaller, meaningful chunks. Traditional SAST struggled to "scale to large applications" in terms of both performance and result management (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). A scan of an entire million-line codebase might produce thousands of findings that were hard to correlate and verify. Cruz's strategy with O2 was to invert this: instead of one monolithic scan, perform targeted scans on MethodStreams that represent key functionality slices (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). This is akin to a divide-and-conquer approach. Each MethodStream file can be scanned individually (either manually or with an automated engine), yielding findings specific to that flow. Since the code in a MethodStream is already connected, the scanner doesn't have to work as hard to infer cross-file links – the links are explicit. This **improves accuracy and reduces false negatives**. It also helps with false positives, because when you review a finding, you have the full context in the MethodStream to judge whether it's exploitable or a false alarm. Scalability is achieved because you can distribute the analysis: for example, 100 web service endpoints become 100 MethodStream files that could even be analyzed in parallel. O2 also integrated with other analysis engines – for instance, running Microsoft's Cat.NET analyzer on a MethodStream, or using Fortify/Checkmarx rules on the consolidated code – leveraging those tools' strengths on the focused input (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'). The result is that issues which would be "diluted" in a giant code scan bubble up clearly in a targeted scan. Another advantage for large projects is

maintainability of results: if the code changes, only the affected MethodStreams need to be regenerated and re-scanned, rather than rescanning everything. This approach presaged how modern pipelines might do incremental scanning. Importantly, Cruz's emphasis on **"trace connection"** (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size') is a lesson in scale: a huge application often can't be fully understood in one go, but if you can connect the pieces, you can still achieve comprehensive coverage. By focusing on the connectivity of data flows (and automating that via MethodStreams), O2 allowed security testers to handle big apps without drowning in noise or missing critical interactions. This methodology encourages contemporary SAST users to think in terms of flows/features rather than files – an approach that has proven effective in large-scale code security reviews.

5. The Role of GenAI

The landscape of application security is changing with the advent of Generative AI (GenAI) technologies, and many of the challenges that MethodStreams aimed to solve can now be tackled in new ways. Dinis Cruz has pointed out that earlier attempts at static analysis were often limited – *"ridiculous static analysis that just about didn't work,"* as he describes some past tools (Cybersecurity in the GenAI age with Dinis Cruz from The Cyber Boardroom (Practical AI #286)) – especially when it came to asking high-level security questions of an application. GenAI (such as large language models like GPT-4) offers a path to address these limitations by bringing **scalability, explainability, and automation** to SAST.

Scalability: GenAI models can ingest and reason about large codebases in a more flexible manner than rigid static analyzers. In the past, making an engine handle millions of lines and complex inter-module traces was extremely hard, as evidenced by the need for techniques like MethodStreams. Now, with AI, we can prompt a model with *specific queries* or incremental chunks of code, and it can generalize or "connect the dots" across those pieces. Cruz noted that we can start to **"codify a lot of those [analysis] intents in simple language, instead of code"** (Cybersecurity in the GenAI age with Dinis Cruz from The Cyber Boardroom (Practical AI #286)). For example, rather than writing a custom dataflow analyzer, we might ask the AI, *"Does this web service method ever allow unvalidated input to reach a database query?"* The AI can interpret this in plain English and attempt to trace that condition through the code. This natural language approach means we can scale security analysis to cover logic and business rules that would be very difficult to hard-code into a traditional tool. Additionally, AI can handle a breadth of data – configuration files, documentation, and code – simultaneously, which is important for scaling to real-world app complexity. Cruz suggests that with GenAI we can analyze the *intent* behind code changes or features and get a *"better sense of what is happening"* in the application (Cybersecurity in the GenAI age with Dinis Cruz from The Cyber Boardroom (Practical AI #286)), something that scales the security review beyond just pattern matching and into understanding.

Explainability: One of the powerful aspects of GenAI in the context of SAST is its ability to produce human-readable explanations. MethodStreams made vulnerabilities more explainable by providing all the code in one place; GenAI can take it further by summarizing and describing the vulnerability in natural language. For instance, after identifying a potential SQL injection path, an AI could generate a narrative: "**The user input from function X reaches the database in function Y without sanitization, allowing an attacker to inject SQL**." This is essentially an automated report-writing capability. In discussions on AI and security, Cruz highlighted that AI can *"help explain applications"* and security impacts in ways developers can easily grasp (Cybersecurity in the GenAI age with Dinis Cruz from The Cyber Boardroom (Practical AI #286)). While MethodStreams provided the raw material for explanation (the code), an AI assistant could analyze that material and produce an explanation or even suggest a fix. This greatly aids in **developer-facing security** – an area that O2 also aimed to improve by integrating with IDEs for real-time feedback. Moreover, GenAI's explainability helps in triaging findings. It can reason about whether a detected data flow is truly dangerous or if some mitigating factor exists, and then articulate that reasoning. This addresses the age-old problem of static analysis results being too cryptic or overwhelmed by false positives. With AI, each finding can come with a rationale in plain language, making the results more actionable and trustworthy. In short, GenAI brings a level of **contextual understanding and communication** that complements the pure code-centric view of traditional MethodStreams.

Automation: Perhaps the most exciting role of GenAI is in automating the heavy lifting that previously required custom code like the O2 Platform. Creating MethodStreams was itself an automated refactoring task – one that required a deep understanding of the code structure. Today, a sufficiently advanced AI could potentially generate a MethodStream on the fly: given a starting function, it could identify relevant code and even produce a merged excerpt for the user, without the user writing a dedicated tool for it. In practice, AI coding assistants could write the code to produce MethodStreams or CodeStreams in various languages, effectively democratizing the approach. Cruz has argued that *"with GenAI, we can finally make AppSec work"* at scale, implying that tasks which were once too time-consuming to script (or too brittle) can now be handled by AI-driven tools (It's 2024 And, With GenAI, We Can Finally Make AppSec Work). This includes automating security testing workflows end-to-end. For example, an AI could detect a dangerous flow, explain it, and even create a unit test or exploit script to prove it – all automatically. We already see early signs of this in tools that integrate AI for security code review. GenAI also excels at pattern recognition, which means it can detect variants of vulnerabilities across a large codebase faster than a human crafting MethodStreams for each entry point. It can act like a smart SAST engine that learns what a vulnerability looks like in context and searches for similar patterns elsewhere, something Cruz alluded to when he mentioned analyzing the application in a more "three-dimensional" way (Cybersecurity in the GenAI age with Dinis Cruz from The Cyber Boardroom (Practical AI #286)). Finally, AI's ability to interface with humans via natural language means automation is more accessible. A developer or tester can *ask* the AI to perform a security analysis task (e.g., "find any unvalidated inputs in this repository that reach file system calls") and the AI will attempt it. This removes friction and required expertise, encouraging more frequent and thorough security checks throughout development. In essence, GenAI can be seen as a layer on top of concepts like MethodStreams

– not replacing them, but using them internally and presenting results in a smarter way. Where MethodStreams solved a problem by restructuring code, GenAI can solve it by reasoning and language, guided by training on vast amounts of code and vulnerabilities.

6. Conclusion and Future Work

Method Streams and Code Streams, as pioneered in Dinis Cruz's O2 Platform, represent a significant advance in static analysis techniques for security. They transformed the daunting task of tracing vulnerabilities in large, complex applications into a more manageable form by **automating trace connection**. In this paper, we reviewed how MethodStreams consolidate an entire call tree's code into one stream, enabling both tools and humans to easily "follow the taint" from sources to sinks (Owasp o2 platform november 2010 | PPT). We also saw how CodeStreams build on this by extracting individual data flow paths, effectively performing taint analysis to expose specific vulnerability traces (Read Generation Z Developers | Leanpub). Technically, the implementation leveraged AST parsing and code refactoring, showcasing a clever use of compiler technology to aid security reviews. The result was improved vulnerability detection – as evidenced by case studies like HacmeBank – and a more scalable approach to SAST for large codebases. Instead of asking "can we scan millions of lines?", Cruz's approach taught us to ask "can we connect the traces?" (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'), which is ultimately more important for finding real issues.

The key findings highlight the importance of **context in static analysis**: by providing context (in the form of all relevant code) MethodStreams made static analysis smarter without changing the underlying analysis engine. This idea is incredibly important even today. Modern development practices, with microservices and cloud functions, still suffer from trace fragmentation – and the MethodStreams concept is a strategy to counter that. We encourage the adoption of similar techniques in contemporary tools and environments. For instance, SAST vendors could incorporate an option to auto-generate a consolidated view of each data flow or use graph databases to achieve a similar effect. Likewise, developers can use open-source AST libraries to script their own custom "stream" views for particularly security-critical parts of their systems. As Cruz advised future generations of developers, *this type of AST manipulation is a valuable area of research to focus on* (Read Generation Z Developers | Leanpub) – it equips one with powerful capabilities to inspect and transform code. Re-implementing the MethodStreams/CodeStreams approach in modern languages (be it using Python AST, JavaScript Babel AST, or .NET Roslyn) is a worthwhile effort to bring these benefits to today's codebases.

Looking forward, the integration of GenAI with concepts like MethodStreams appears especially promising. Where O2 required an expert to set up the analysis and interpret it, an AI-augmented system could generate streams or perform similar trace consolidations automatically and even explain them in plain language. This could finally realize at scale the vision that guided O2: an *"Application Visibility" engine that makes security knowledge accessible to all* (Dinis Cruz Blog: OWASP O2 Platform). In 2024 and beyond, as Cruz has noted, we have an opportunity with GenAI to **"finally make AppSec work"** in ways that were not possible before (It's 2024

And, With GenAI, We Can Finally Make AppSec Work). The foundational ideas from O2 Platform – automate what the expert does, integrate security deeply into the development workflow, and focus on traceability – remain highly relevant. By adopting those ideas and enhancing them with modern AI and tooling, the security community can create smarter static analysis tools that not only find more vulnerabilities, but do so in a developer-friendly and scalable manner. The work on MethodStreams and CodeStreams stands as an early example of this philosophy, and its revival in modern projects will be a fitting continuation of Dinis Cruz's legacy in application security innovation.

References: Dinis Cruz's blogs, presentations, and writings on the O2 Platform and AST techniques were used as the primary sources for this paper, including content from his personal blog (Dinis Cruz Blog: O2 .NET SAST Engine: MethodStream and CodeStrams for a WebService Method) (Dinis Cruz Blog: In SAST the issue is 'Trace Connection', not 'Scan Size'), the Generation Z Developers book (Read Generation Z Developers | Leanpub) (Read Generation Z Developers | Leanpub), and OWASP conference materials (Owasp o2 platform november 2010 | PPT). All examples and explanations are drawn from these published works to ensure accuracy and authenticity.