CROW: Code Diversification for WebAssembly

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Orestis Floros
KTH Royal Institute of Technology
Stockholm, Sweden
forestis@kth.se

Oscar Luis Vera Perez
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
oscar.vera-perez@inria.fr

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

***Abstract*—The adoption of WebAssembly increases rapidly, as it provides a fast and safe model for program execution in the browser. However, WebAssembly is not exempt from vulnerabilities that can be exploited by malicious observers. Code diversification can mitigate some of these attacks. In this paper, we present the first fully automated workflow for the diversification of WebAssembly binaries. We present CROW, an open-source tool implementing this workflow through enumerative synthesis of diverse code snippets expressed in the LLVM intermediate representation. We evaluate CROW's capabilities on 303 C programs and study its use on a real-life security-sensitive program: libsodium, a modern cryptographic library. Overall, CROW is able to generate diverse variants for 239 out of 303 (79%) small programs. Furthermore, our experiments show that our approach and tool is able to successfully diversify off-the-shelf cryptographic software (libsodium).**

I. Introduction

WebAssembly is the fourth official language of the Web [36]. The language provides low-level constructs enabling efficient execution times, much closer to native code than JavaScript. It constitutes a fast and safe platform to execute programs in the browser and embedded environments [21]. Consequently, the adoption of WebAssembly has been rapidly growing since its introduction in 2015. Nowadays, languages such as Rust and C/C++ can be compiled to WebAssembly using mature toolchains and can be executed in all notable browsers.

The WebAssembly execution model is designed to be secure and to prevent many memory and control flow attacks. Still, as its official documentation admits [11], WebAssembly is not exempt from vulnerabilities that could be exploited [30]. Code diversification [5], [28] is one additional protection that can harden the WebAssembly stack. This consists in synthesizing different variants of an original program that provide the same functionalities but exhibit different execution traces. In this paper, we investigate the feasibility of diversifying WebAssembly code, which is, to the best of our knowledge, an unresearched area.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/madweb.2021.23xxx
www.ndss-symposium.org

Our contribution is a workflow and a tool, called CROW, for automatic diversification of WebAssembly programs. It takes as input a C/C++ program and produces a set of diverse WebAssembly binaries as output. The workflow is based on enumerative code synthesis. First, CROW lists blocks that are potentially relevant for diversification, second, CROW enumerates alternative instruction sequences, and third, CROW checks that the new instruction sequences are functionally equivalent to the original block. CROW builds on the idea of superdiversification [25] and extends the concept to the enumeration of a set of variants instead of synthesizing only one solution. We also take into account the specificities of WebAssembly and the details of its execution.

We evaluate the diversification capabilities of CROW in two ways. First, we diversify 303 small C programs compiled to WebAssembly. Second, we run CROW to diversify a real-life cryptographic library that natively supports WebAssembly. In both cases, we measure the diversity among binary code variants, as well as the diversity of execution traces. When measuring the diversity in binary code, we compare the WebAssembly and the machine code variants. This way we assess the ability of CROW at synthesizing variations in WebAssembly, as well as the extent to which these variations are preserved when compiling WebAssembly to machine code. Our original experiments demonstrate the feasibility of diversifying WebAssembly code. CROW generates diverse variants for $239/303\,(79\%)$ C programs. TurboFan, the optimizing compiler used in the V8 engine, preserves 99.48% of these variants. CROW successfully synthesizes variants for the cryptographic library. The variants indeed yield either different execution traces. This is promising milestone in getting a more secure Web environment through diversification.

To sum up, our contributions are:

- CROW: the first automated workflow and tool to diversify WebAssembly programs, it generates many diverse WebAssembly binaries from a single input program.
- A quantitative evaluation over 303 programs showing the capability of CROW to diversify WebAssembly binaries and measuring the impact of diversification on execution traces.
- A feasibility study of the diversification on a real-world WebAssembly program, demonstrating that CROW can handle libsodium, a state-of-the-art cryptographic library.

II. Background

A. WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. It is designed to address the problem of safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware or languages and can run in a standalone Virtual Machine (VM) or in other environments such as Arduino [20]. A paper by Haas et al. [21] formalizes the language and its type system, and explains the design rationale.

Listing 1 and 2 illustrate WebAssembly. Listing 1 presents the C code of two functions and Listing 2 shows the result of compiling these two functions into a WebAssembly module. The `type` directives at the top of the module declare the function: the types of its parameters and the type of the result. Then, the definitions for the function follow. These definitions are sequences of stack machine instructions. At the end, the `main` function is exported so that it can be called from outside this WebAssembly module, typically from JavaScript. WebAssembly has four primitive types: integers (`i32` and `i64`) and floats (`f32` and `f64`) and it includes structured instructions such as `block`, `loop` and `if`.

Listing 1: C function that calculates the quantity $2x + x$

```
int f(int x) { return 2 * x + x; }

int main(void) { return f(10); }
```

Listing 2: WebAssembly code for Listing 1.

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;1;) (func (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
  (func (;1;) (type 1) (result i32)
    i32.const 10
    call 0)
  (export "main" (func 1)))
```

WebAssembly is characterized by an extensive security model [11] founded on a sandboxed execution environment that provides protection against common security issues such as data corruption, code injection and return oriented programming (ROP). However, WebAssembly is no silver bullet and is vulnerable under certain conditions [30]. This motivates our work on software diversification as one possible mitigation among the wide range of security counter-measures.

B. Motivation for Moving Target Defense in the Web

The distribution model for web computing is as follows: build one binary and distribute millions of copies, all over the world, which run on browsers. In this model an attacker has two key advantages over the developers: she has a runtime environment that she fully controls and observes in any possible way. Consequently, when she finds a flaw in this virtually transparent environment, knowing that this flaw is present in the millions of copies that have been distributed over the world, she can exploit the flaw at scale.

The developers can never assume that they can control the web browser. Yet, they can challenge the second advantage of the attacker, known as the break-once-break-everywhere advantage. The developers can stop distributing clones of the binary and distribute diverse versions instead, as suggested by the pioneering software diversification works of Cohen [12] and Forrest et al. [19].

In the context of diversification, moving target defense [40] means distributing diverse variants constantly. In the context of the web, it means distributing a different variant at each HTTP request. Moving target defense is appropriate for mitigating yet unknown vulnerabilities. The diversification technique does not always remove the potential flaws, yet the vulnerabilities in the diversified binaries can be located in different places. With moving target defense, a successful attack on one browser cannot be performed on another browser with the same effectiveness. The diversified binaries that CROW outputs can be used interchangeably over the network, in a moving target defence choreographed over the web.

To sum up, by combining moving target defense deployment to diversification, we reduce the information asymmetry between the Web attacker and the defender, increasing the uncertainty and complexity of successful attacks over all client browsers [16], [42].

III. CROW's Diversification Technique

In this section we describe the workflow of CROW for diversifying WebAssembly programs. First we introduce the main concepts behind CROW. Then, we describe each stage of the workflow and we discuss the key implementation details.

A. Definitions

In this subsection we define the key concepts for CROW.

Definition 1: Block (based on Aho et al. [2]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.

Definition 2: Program state (based on Mangpo et al. [35]): At any point in time, the program state S is defined as the collection of local and global variables, and, the program counter pointing to the next instruction.

Definition 3: Pure block: A block B is said to be pure if and only if, given the program state S_i, every execution of B produces the same state S_o.

Definition 4: Functional equivalence modulo program state (based on Le et al. [29]): Let B_1 and B_2 be two blocks. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.

Definition 5: Code replacement: Let P be a program and T a pair of blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both pure as defined in Definition 3 and functionally equivalent as defined in Definition 4. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.

CROW generates new program variants by finding and applying code replacements as defined in Definition 5. A program variant could be produced by applying more than one candidate code replacement. For example, the tuple, composed by the code blocks in Listing 3 and Listing 4, is a code replacement for Listing 2.

Listing 3: WebAssembly pure code block from Listing 2.

```
local.get 0
i32.const 2
i32.mul ; 2 * x ;
```

Listing 4: Code block that is functionally equivalent to Listing 3

```
local.get 0
i32.const 1
i32.shl ; x << 1 ;
```

B. Overview

CROW synthesizes variants for WebAssembly programs. We assume that the programs are generated through the LLVM compilation pipeline. This assumption is motivated as follows: first, LLVM-based compilers are the most popular compilers to build WebAssembly programs [30]; second, the availability of source code (typically C/C++ for WebAssembly) provides a structure to perform code analysis and produce code replacements that is richer than the binary code.

CROW takes as input a C/C++ program and produces a set of unique, diversified WebAssembly binaries. Figure 1 shows the stages of this workflow. The workflow starts with compiling the input program into LLVM bitcode using clang. Then, CROW analyzes the bitcode to identify all pure blocks and to synthesize a set of candidate replacements for each pure block. This is what we call the *exploration* stage. In the *generation* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants. Finally, those bitcode variants are compiled to WebAssembly binaries that can be sent to web browsers.

Challenges. The concept of diversifying WebAssembly programs is novel and it is arguably hard for the following reasons. First, WebAssembly is a structured binary format, without goto-like instructions. This prevents the direct application of a wide range of diversification operators based on goto [41]. Second, the existing transformation and diversification tools target instruction sets larger than the one of WebAssembly [39]. This limits the efficiency of diversification, and the possibility of searching for a large number of equivalent code replacements. We address the former challenge using the LLVM intermediate representation as the target for diversification. We address the latter challenge by tailoring a superoptimizer for LLVM, using its subset of the LLVM intermediate representation. In particular, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly (for example, `freeze` instructions), which is an essential step to get executable diversified WebAssembly code.

C. Exploration stage

Given a program P for which we want to generate WebAssembly variants, the exploration stage of CROW identifies all pure blocks in the LLVM bitcode of P. CROW considers every directed acyclic graph contained in one function as a pure block. Then, CROW searches for code replacements for each one of them.

The generation of a code replacement consists of two steps. First, the synthesis of the new block, and, second, equivalence checking. Every variant block that passes the equivalence check is stored for use in diversification. The synthesis of block variants consists of enumerating all possible blocks that can be built as a combination of a given number of instructions, bounded by a maximum value to keep a tractable synthesis space.

There are two parameters to control the size of the search space and hence the time required to traverse it. On one hand, one can limit the size of the variants. In our experiments we limit the block variants to a maximum of 50 instructions. On the other hand, one can limit the set of instructions that are used for the synthesis. In our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer). This configuration allows the user to find a trade-off between the amount of variants that are synthesized and the time taken to produce them.

Listing 5: Listing 1 in LLVM's intermediate representation.

```
define i32 @f(i32) {
 %2 = mul nsw i32 %0,2
 %3 = add nsw i32 %0,%2

 ret i32 %3
}

define i32 @main() {
 %1 = tail call i32 @f(i32 10)
 ret i32 %1
}
```

Block A

```
%2 = mul nsw i32 %0,2
```

Block B

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2
```

In Listing 5 we illustrate the LLVM bitcode representation of Listing 1. In this bitcode, CROW identifies two pure blocks in function `f()`, which are displayed on the right part of the listing, in gray and green. The first pure block is composed of one single instruction (line 2) that performs the `2*x` multiplication. The second block has two instructions, one multiplication and one addition.

Using CROW, it is possible to diversify both blocks. For example, using a maximum of 1 instruction per replacement and searching over the complete bitcode instruction set, a potential replacement for Block A is: `%2 = shl nsw i32 %0,1 %`. This replacement calculates the same expression `2*x`, using a shift left operation.

To determine the equivalence between a pure block and a candidate replacement, we use an equivalence checker based on SMT [17]. In our example, the checker would prove that there cannot be a value of x such that $2 * x \neq x \ll 1$. In general, if no such counter-example exists, then the functional equivalence is assumed. On the other hand, if there exists an input resulting in different outputs for a block and a variant, then they are proven not equivalent and the variant is discarded.

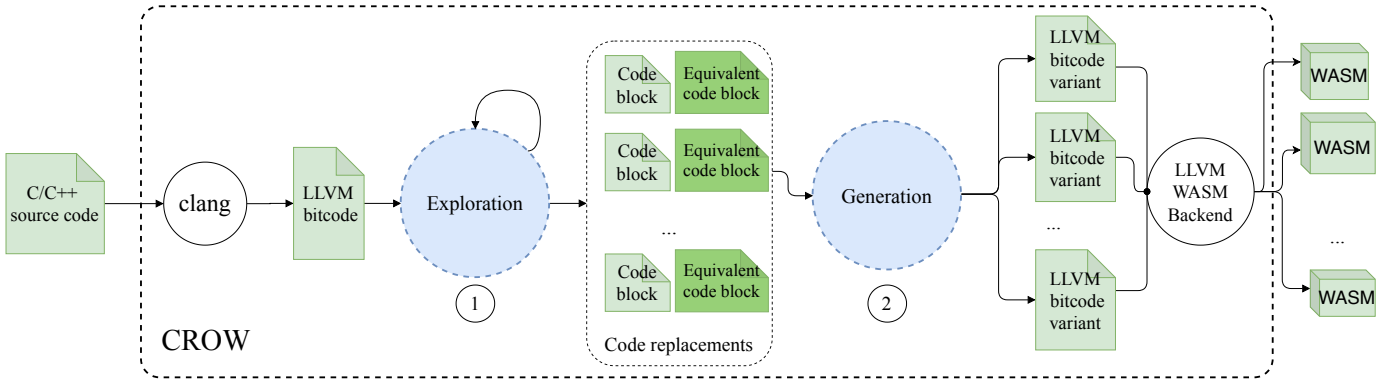

Fig. 1: CROW's workflow for diversifying WebAssembly programs.

D. Generation stage

In this stage, we select and combine code replacements that have been synthesized during the exploration stage, in order to generate WebAssembly binary variants. We apply each code replacement to the original program to produce a LLVM IR variant. Then, this IR is compiled into a WebAssembly binary. CROW generates WebAssembly binaries from all possible combinations of code replacements as the power set over all code replacements.

After the exploration phase, it is possible that two subsets of code replacements overlap, *i.e.,* they produce the same WebAssembly binary. The overlap between blocks is explained as follows: Let $S = \{(B_1, R_1), (B_1, R_2), \cdots, (B_n, R_m)\}$ be a set of candidate replacements over a program P. If two blocks from the original program B_i, B_j, $j \neq i$, overlap, *i.e.,* the intersection of $\mathrm{CFG}(B_i)$[1] and $\mathrm{CFG}(B_j)$ is not empty, then only the replacements of the largest original block are preserved when combining blocks.

In this example, the exploration stage synthesizes $6 + 1$ bitcode variants for the considered blocks respectively, which results in 14 module variants (the power set combination size). Yet, the generation stage would eventually generate 7 variants from the original WebAssembly binary. This gap between the number of potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composing several variants into one.

E. Implementation

The majority of the WebAssembly applications are built from C/C++ source code using the LLVM toolchain. Consequently, the implementation of CROW is based on LVVM/ Furthermore, CROW extends Souper [38], a superoptimizer for LLVM that aims to reduce the size of binary code. Souper has its own intermediate representation, which is a subset of the LLVM IR.

To extract code blocks, we scan LLVM modules, looking for instructions that return integer-typed values. Each such instruction is considered as the exit of a code block. Souper's representation of a code block is built as a backward traversal process through the dependencies of the detected instruction. If memory loads or function calls are found, the backward traversal process is stopped and the current instruction is considered as an input variable for the code block. Notice that, by construction, Souper's translation is oblivious to the memory model, thus, it cannot infer string data types or other abstract data types. The translation from Souper IR to a BitVector SMT theory is done on the fly. Souper uses the z3[2] solver to check the equivalence between a code block original and a potential replacement for it.

We now summarize the main changes that we implement in Souper and in the LLVM backend in order to support diversification. Souper, as a superoptimizer, aims at generating a single variant that is smaller than the original, yet we want to obtain as many blocks as possible. To achieve automatic diversification, we modify Souper to disable the key cost restriction functions, data-flow pruning and peephole optimizations, all being detrimental for diversification. In order to increase the number of variants that CROW can generate, CROW parallelizes the process of replacement synthesis.

In addition, CROW orchestrates a series of Souper executions with various configurations (in particular the size of the replaced expression). Finally, we carefully fine-tune a set of 19 Souper options to ensure that the search is effective for diversification in feasible time.

In the generation stage of CROW, we also modify Souper to amplify the generation of WebAssembly binary diversity. Initially, Souper generates a single bitcode variant, inserting all replacements at once. We modify it so that we can obtain a combination of code replacements. Finally, on the LLVM side, we disable all peephole optimizations in the WebAssembly backend, in particular instructions merging and constant folding. This aims to preserve the variations introduced in the LLVM bitcode during the generation of binaries.

The implementation of CROW is publicly available for sake of open science and can be reviewed at https://github.com/KTH/slumps/tree/master/crow.

IV. Evaluation Protocol

To evaluate the capabilities of CROW to diversify WebAssembly programs, we formulate the following research

[1] $\mathrm{CFG}(A)$ refers to backward Control Flow Graph starting at inst. A.

[2] https://github.com/Z3Prover/z3

questions:

RQ1: **To what extent are the program variants generated by CROW statically different?** We check whether the WebAssembly binary variants produced by CROW are different from the original WebAssembly binary. Then, we assess whether the generation of x86 machine code performed by V8's WebAssembly engine preserves CROW's transformations.

RQ2: **To what extent are the program variants generated by CROW dynamically different?** It is known that not all diversified programs produce distinguishable executions [15], sometimes it is impossible to observe different behaviors between variants. We check for the presence of different behaviors with a custom WebAssembly interpreter, characterizing the behavior of a WebAssembly program by its stack operation trace.

RQ3: **To what extent can CROW be applied to diversify real-world security-sensitive software?** We assess the ability of CROW to diversify a state-of-the-art cryptographic library for WebAssembly, libsodium [18].

A. Corpus

We answer RQ1 and RQ2 with a corpus of programs appropriate for our experiments. We take programs from the Rosetta Code project[3]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains a wide range of tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. Next, we apply a number of filters. We discard 1) all programs that do not compile with `clang`, 2) all interactive programs requiring input from users *i.e.,* invoking functions like `scanf`, 3) all programs that contain more than 100 blocks, 4) all programs without termination, 5) all programs with non-deterministic operations, for example, programs working with time or random functions. This filter produces a final set of 303 programs.

The result is a corpus of 303 C programs. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

B. Protocol for RQ1

With RQ1, we assess the ability of CROW to generate WebAssembly binaries that are different from the original program. For this, we compute a distance metric between the original WebAssembly binary and each binary generated by CROW. Since WebAssembly binaries are further transformed into machine code before they execute, we also check that this additional transformation preserves the difference introduces by CROW in the WebAssembly binary. We use the Turbofan ahead-of-time compiler of V8, with all its possible optimizations, to generate a x86 binary for each WebAssembly binary. Then, we compare the x86 version of each variant against the x86 binary corresponding to the original WebAssembly binary.

We compare the WebAssembly and machine code of each program and its variant using Dynamic Time Warping (DTW) [31]. DTW computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is actually a distance metric, called DTW. The larger the DTW distance, the more different the two sequences are. In our case, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain two DTW distance values for each program-variant pair: one at the level of WebAssembly code and the another one at the level of x86 code. Metric 1 below defines these metrics.

Metric 1: dt_static: Given two programs P_X and V_X written in X code, $dt\_static(P_X, V_X)$, computes the DTW distance between the corresponding program instructions for representation X ($X \in \{Wasm, x86\}$). A $dt\_static(P_X, V_X)$ of 0 means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.

We run CROW on our corpus of 303 programs. We configure CROW to run with a diversification timeout of 6 hours per program. For each program, we collect the set of generated variants. For all pairs program, variant that are different, we compute both dt_static for WebAssembly and x86 representations.

The key property we consider is as follows: if $dt\_static(P_{Wasm}, P'_{Wasm}) > 0$ and $dt\_static(P_{x86}, P'_{x86}) > 0$, this means that both programs are still different when compiled to machine code, and we conclude that V8's compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program (including the original).

C. Protocol for RQ2

For RQ2, we compare the executions of a program and its variants for a given input. In this experiment, we characterize the execution of a WebAssembly binary according to its trace of stack operations.

This method of tracing allows us to evaluate CROW's effect on program execution according to the WebAssembly specification, independently of any specific engine.

For each execution of a WebAssembly program, we collect a trace of stack operations. These traces are composed of stack-type instructions: `push <value>` and `pop <value>`. All traces are ordered with respect to the timestamp of the events. We compare the traces of the original program against those of the variants with DTW. DTW computes the global alignment between two traces and provides a value for the cost of this alignment.

Metric 2: dt_dyn: Given a program P and a CROW generated variant P', dt_dyn(P,P'), computes the DTW distance between the corresponding stack operation traces collected during their execution. A dt_dyn of 0 means that both traces are identical. The higher the value, the more different the stack operation traces.

To answer RQ2 we compute Metric 2 for a study subject program and all the unique program variants generated by CROW in a pairwise comparison. The pairwise comparison

[3]http://www.rosettacode.org/wiki/Rosetta_Code

allows us to compare the diversity between variants as well. We use SWAM[4] to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [10].

The builtin WebAssembly API for JavaScript is usually mutable, thus, the same model for traces collection can be implemented on top of V8. In other words, a custom interpreter can be implemented in order to collect the traces in the browser or standalone JavaScript engines. This validates the usage of SWAM to study the traces diversity.

D. Protocol for RQ3

In RQ3, we assess the ability of CROW to diversify a mature and complex software library related to security. We choose the libsodium [18] cryptographic library, which natively compiles to WebAssembly. With 3752 commits contributed by 96 developers, its API provides the basic blocks for encryption, decryption, signatures and password hashing. We experiment with code revision 2b5f8f2b, which contains 45232 lines of C code. Libsodium has 102 separate WebAssembly modules that we use as input for CROW. Each module corresponds to one C file that encompasses a set of related functions.

To answer RQ3, we run CROW on the libsodium bitcodes, generating a set of WebAssembly variants. Then, we assess both binary code diversity and behavioural diversity between the variants and the original libsodium, using the same techniques as in RQ1 and RQ2.

Collecting traces The libsodium repository includes an extensive test suite of 77 tests, where one test is one usage scenario. We use this test suite to measure the trace diversity among program variants. Since some test traces are larger than 1 GB each, we focus on reasonably sized tests: we select the $41/77$ test cases that produce a trace containing less than 50 million events each.

To measure the relative trace diversification for each test, we normalize the dt_dyn used in RQ2 by dividing it with the length of the original trace. This allows us to compare the relative success of CROW's diversification technique across different tests.

Since libsodium uses a pseudo-number generator, we set a static seed when executing libsodium, so that the diversity observed in traces is only due to CROW's diversification. This seed is given to the `arc4random` API used by libsodium in WebAssembly. To quantify the effectiveness of our diversification technique, we compare the trace distance produced by our technique with the trace distance that occurs when the seed is changed (baseline).

V. Experimental Results

In this section we present the results for the research questions formulated in section IV.

[4]https://github.com/satabin/swam

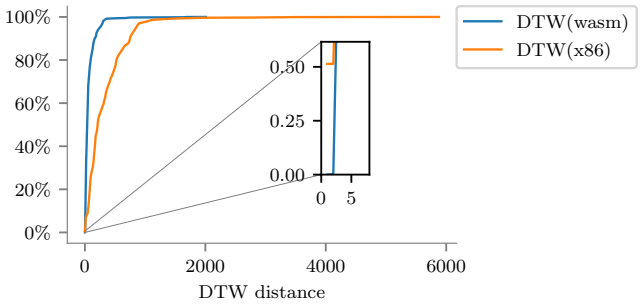

Fig. 2: Cumulative distribution for all pairwise comparisons between a program and its variants. Each line corresponds to a different program representation.

A. To what extent are the program variants generated by CROW statically different?

We run CROW on 303 C programs compiled to WebAssembly. CROW produces at least one unique program variant for $239/303$ programs. For the rest of the programs $(64/303)$, the timeout is reached before CROW can find any valid variant.

We subsequently perform a manual analysis of the programs that yield more than 100 unique WebAssembly variants. This reveals one key reason that favors a large number of unique WebAssembly variants: the programs include bounded loops. In these cases CROW synthesizes variants for the loops by unrolling them. Every time a loop is unrolled, the loop body is copied and moved as part of the outer scope of the loop. This creates a new, statically different, program. The number of programs grows exponentially with nested loops.

A second key factor for the synthesis of many variants relates to the presence of arithmetic. Souper, the synthesis engine used by CROW, is effective in replacing arithmetic instructions by equivalent instructions that lead to the same result. For example, CROW generates unique variants by replacing multiplications with additions or shift left instructions (Listing 8). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 9).

Listing 8: Diversification through arithmetic expression replacement.

```
local.get 0          local.get 0
i32.const 2          i32.const 1
i32.mul              i32.shl
```

Listing 9: Diversification through inversion of comparison operations.

```
local.get 0          i32.const 11
i32.const 10         local.get 0
i32.gt_s             i32.le_s
```

We now discuss the prevalence of the transformations made by CROW when the WebAssembly binaries are transformed to machine code, specifically with the V8's engine. In Figure 2 we plot the cumulative distribution of dt_static, comparing WebAssembly binaries (in blue) and x86 binaries (in orange). The figure plots a total of 103003 dt_static values for each representation, two values for each variant pair comparison (including original) for the 239 program. The value on the y-axis shows which percentage of the total comparisons lie

Listing 10: Excerpt of WebAssembly program p74: CROW replaces a loop by a constant.

```
local.set 1
loop ;; label = @1
  ...
end
...
i32.store
```

```
local.get 0
i32.const 25264
...
i32.store
```

below the corresponding dt_static value on the x-axis. Since we measure the distances between original programs and WebAssembly variants, then 100% of these binaries have $dt\_static > 0$. Let us consider the x86 variants: dt_static is strictly positive for 99.48% of variants. In all these cases, the V8 compilation phase does not undo the CROW diversification transformations. Also, we see that there is a gap between both distributions, the main reason is the natural inflation of machine code. For example, two variants that differ by one single instruction in WebAssembly, can be translated to machine code where the difference is increased by more than one machine code instruction.

The zoomed subplot focuses on the beginning of the distribution, it shows that the dt_static is zero for 0.52% of the x86 binaries. In these cases the V8 TurboFan compiler from WebAssembly to x86 reverts the CROW transformations. We find that CROW produces at least one of these reversible transformations for $34/239$ programs. Listing 11 shows one of the most common transformations that is reversed by TurboFan, according to our experiments.

Listing 11: Replacement in WebAssembly that is translated to the same x86 code by V8-TurboFan.

```
i32.const -<n>
i32.sub
```

```
i32.const <n>
i32.add
```

We look at the cases that yield a small number of variants. There is no direct correlation between the number of identified blocks and the number of unique variants. We manually analyze programs that include a significant number of pure blocks, for which CROW generates few variants. We identify two main challenges for diversification.

1) Constant computation We have observed that Souper searches for a constant replacement for more than 45% of the blocks of each program while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because CROW is oblivious to a memory model.

2) Combination computation The overlap between code replacements, discussed in subsection III-D, is a second factor that limits the number of unique variants. CROW can generate a high number of variants, but not all replacement combinations are necessarily unique.

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the 239 programs with their variants. The ratio of size change between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for all Rosetta programs. This limited impact on the binary size of the variants is good news because they are meant to be distributed to browsers over the network.

Answer to RQ1

CROW is able to generate diverse variants of WebAssembly programs for $239/303\,(79\%)$ programs in our corpus. We observe that programs that include bounded loops and arithmetic expressions are highly prone to diversification. V8's TurboFan compilation to x86 code preserves 99.48% of the transformations performed by CROW. To our knowledge, this is the first ever realization of automated diversification for WebAssembly.

B. To what extent are the program variants generated by CROW dynamically different?

Now, we focus on the 41 programs that have at least 9 unique WebAssembly variants in order to study the diversity of execution traces. We apply the protocol described in subsection IV-C by executing the WebAssembly programs and their unique variants in order to collect the stack operation traces. Then, we compare the traces of each pair of original program and a variant. We run 1906 program executions and we perform 98774 trace pair comparisons.

Table I summarizes the observed trace diversity, as captured by dt_dyn (Metric 2), among each program and their variants. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. The table summarizes the distribution of distances between stack operation trace pairs: the minimum value, the maximum value, the median value, the percentage of values equal to zero and the percentage of values greater than zero. The programs are sorted with respect to the number of unique variants. The green highlight color in $> 0\%$ columns represents more than 50% of non-zero comparisons, *i.e.,* high diversification. For instance, the first row shows the trace diversity for `p96`, where 99.70% of the pairwise comparisons between all collected traces have a different dt_dyn.

For the stack operation traces, all programs have at least one variant that produces a trace different from the original. All but one (`p81`) programs have the majority of variants producing a different stack operation trace. This shows the real effectiveness of CROW for diversifying stack operation traces.

We manually analyze variants with high and low trace diversity. We observe that constant inferring is effective at changing the stack operation trace. For instance, for program `p74` shown in Listing 10, CROW removes a loop by replacing it with a constant assignment. The execution of this variant produces traces that are different because the loop pattern is not visible anymore in the trace, and consequently, the distance between the original and the variant traces is large.

	NAME	#var	Σ	Min	Max	Median	0 %	> 0 %
1	**p96**	220	15	0	24062	820	0.30	99.70
2	p56	192	36	0	45420	1416	1.84	98.16
3	p78	159	35	0	20501	759	1.52	98.48
4	p111	144	45	0	2114	520	3.74	96.26
5	**p166**	101	152	0	44538	66	45.80	54.20
6	p122	91	34	0	46026	6434	0.24	99.76
7	p67	89	77	0	94036	85692	0.29	99.71
8	p68	85	10	0	10554	260	3.64	96.36
9	p80	78	9	0	17238	618	3.92	96.08
10	p204	77	42	0	36428	3356	0.33	99.67
11	p183	76	9	0	90628	84402	0.57	99.43
12	p136	62	70	0	62953	58028	0.60	99.40
13	p167	46	232	8	888	724	0.00	100.00
14	p226	42	13	0	90736	74476	8.26	91.74
15	p99	38	74	16	9936	5037	0.00	100.00
16	p18	36	7	0	15620	145	1.10	98.90
17	**p140**	29	17	0	13280	172	6.59	93.41
18	p59	27	6	0	85390	40	1.43	98.57
19	p199	21	87	0	27482	728	4.68	95.32
20	**p91**	21	21	0	50002	228	43.81	56.19
21	p223	21	115	16	40911	632	0.00	100.00
22	p168	20	6	0	22200	18896	2.20	97.80
23	p174	18	40	6	6566	6395	0.00	100.00
24	**p81**	17	86	0	4419	0	84.62	15.38
25	p141	17	6	8	2894	132	0.00	100.00
26	p108	16	6	0	85168	79903	8.97	91.03
27	p98	15	4	0	33	25	6.06	93.94
28	p89	14	45	10	15952	89	0.00	100.00
29	p36	14	52	312	33266	30298	0.00	100.00
30	**p135**	13	5	0	20288	20163	3.57	96.43
31	p161	12	91	240	9792	1056	0.00	100.00
32	p147	12	32	0	54071	21274	7.14	92.86
33	p11	10	38	29798	51846	35119	0.00	100.00
34	p125	10	51	0	4399	4368	7.14	92.86
35	p131	9	4	140	1454	685	0.00	100.00
36	p69	9	48	28	29243	28956	0.00	100.00
37	p134	9	20	4	514	186	0.00	100.00
38	**p74**	9	19	126	8332	6727	0.00	100.00
39	p79	9	97	4	29	16	0.00	100.00
40	p33	9	52	4	2342	15	0.00	100.00
41	p157	9	64	36	242	166	0.00	100.00

TABLE I: Dynamic diversity for 41 diversified WASM programs. The dynamic diversity is captured by dt_dyn between traces. The rows are sorted by the number of unique variants per program. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. Following, the stats for the dt_dyn metric. The colorized cells in the $> 0\%$ column represent high diversification.

Listing 12: Statically different WebAssembly replacements with the same behavior, gray for the original code, green for the replacement.

```
(1) i32.lt_u  i32.lt_s     (3) i32.ne      i32.lt_u
(2) i32.le_s  i32.lt_u     (4) local.get 6 local.get 4
```

We note that there is no relation between the trace distance and the number of block replacements. A high trace distance does not necessarily imply a high number of replacements. For instance, program `p135` has only 4 possible replacements overall its 5 identified blocks yet a median dt_dyn of 20163.

We subsequently analyze the cases where diversification is not reflected in stack operation traces. For example, more than 40% of the pairwise dt_dyn distances for `p166`, `p91` and `p81` are equal to zero. This indicates a lower diversity among the population of variants, than for all the other programs. This happens because some variants have two different bitcode instructions (original and replacement) that trigger the same stack operations. The instructions in Listing 12 are concrete cases of such kind of replacements. The four cases in Listing 12 leave the same value in the stack operation trace. For each case, the original instruction and the replacement are semantically equal in the program domain. The fourth case is a local variable index reallocation, this replacement only changes the index of the local variable but not the event in the stack operation trace. These replacements are sound, produce statically diverse code, but they are not useful to dynamically diversify the original program. This confirms the complementary of using static and dynamic metrics to assess diversification.

The effectiveness of CROW on diversifying stack operation traces is significant. In a security context, such diverse stack operation traces are likely to mitigate potential side-channel attacks [30]. Notably, the attacks based on code profiling are affected when the executed opcodes and the corresponding profiles are different [37].

Answer to RQ2

CROW is successful at generating diverse WebAssembly variant programs, for which we are able to observe different stack operation traces. In other words, CROW generates dynamically different binaries, and ensures that variants of a given program yield different stack operation traces.

C. To what extent can CROW be applied to diversify real-world security-sensitive software?

We run CROW on each of the 102 modules of libsodium with a 6-hour timeout. We find $45/102$ modules that do not contain any pure block, so they are not amenable to our diversification technique. CROW produces at least one valid WebAssembly module variant for 15 of the remaining 57 modules.

Module & Description	#var	#func	Diversified Functions	#calls
argon2-core Core functions for the implementation of the Argon2 key derivation (hash) function [9].	17	6	argon2_finalize argon2_free_instance argon2_initialize	0 0 0
argon2-encoding Functions for encoding and decoding (including salting) Argon2 [9] hash strings.	11	2	argon2_decode_string argon2_encode_string	0 0
blake2b-ref Reference implementation for the BLAKE2 [4] hash function.	7	11	blake2b blake2b_salt_personal blake2b_update	0 1.46E+04 2.04E+04
chacha20_ref Reference implementation of the ChaCha20 stream cipher [6].	7	5	chacha20_encrypt_bytes stream_ietf_ext_ref_xor_ic stream_ref stream_ref_xor_ic	3.51E+06 7.62E+03 1.14E+04 1.14E+05
codecs Implementations of commonly used codecs for conversions between binary formats like Base64 [26].	79	5	sodium_base642bin sodium_base64_encoded_len sodium_bin2base64 sodium_bin2hex sodium_hex2bin	0 0 0 2.57E+05 0
core_ed25519 Implementation of the Edwards-curve Digital Signature Algorithm [8].	2	19	crypto_core_ed25519_is_valid_point	0
crypto_scrypt-common Utility and low-level API functions for the scrypt key derivation (hash) function [34].	5	5	escrypt_gensalt_r	0
pbkdf2-sha256 Implementation of the Password-Based Key Derivation Function 2 (PBKDF2) [27].	14	1	escrypt_PBKDF2_SHA256	0
pwhash_scryptsalsa208sha256 High-level API for the scrypt key derivation function [34].	8	19	crypto_pwhash_scryptsalsa208sha256	0
pwhash_scryptsalsa208sha256_nosse Same as above, but does not use Streaming SIMD Extensions (SSE).	32	3	escrypt_kdf_nosse salsa20_8	0 0
randombytes Pseudorandom number generators.	1	11	randombytes_uniform	5.61E+02
salsa20_ref Contains a reference implementation of the Salsa20 stream cipher [7].	12	2	stream_ref stream_ref_xor_ic	1.14E+04 1.14E+05
scalarmult_ristretto255_ref10 Implementation of the Ristretto255 prime order elliptic curve group [22].	29	4	scalarmult_ristretto255 scalarmult_ristretto255_base scalarmult_ristretto255_scalarbytes	0 0 0
stream_chacha20 High-level API for the ChaCha20 stream cipher [8].	2	15	crypto_stream_chacha20 crypto_stream_chacha20_ietf crypto_stream_chacha20_ietf_ext crypto_stream_chacha20_ietf_ext_xor_ic crypto_stream_chacha20_ietf_xor crypto_stream_chacha20_ietf_xor_ic crypto_stream_chacha20_xor crypto_stream_chacha20_xor_ic	6.65E+02 3.19E+03 2.66E+03 1.68E+02 1.68E+02 2.32E+03 0 1.68E+02
verify Functions used to compare secrets in constant time to avoid timing attacks.	7	6	crypto_verify_16 crypto_verify_32 crypto_verify_64	2.69E+05 3.40E+03 0
Total	**256**	**114**	**40 functions**	

TABLE II: Libsodium modules with at least one variant generated by CROW. The columns on the left include the facts about each module. The first column contains the name and the functional description of the modules. The second column, *#var* (highlighted) gives the number of unique variants generated by CROW. The third column, *#func*, lists the total amount of functions in each module. The remaining columns include a list of functions that CROW has successfully diversified and the number of calls per function in the test suite.

Table II presents the key results for these 15 successfully diversified modules. The first two columns contain the name and description of the diversified module, and, the number of unique static variants. The other columns show the total number of functions inside the module, the names of the diversified functions and the number of calls to each function in the considered tests.

Generation of WebAssembly library variants from WebAssembly module variants. The successfully diversified modules can be combined to obtain a large pool of different versions of the packaged libsodium WebAssembly library. The Cartesian product of all module variants produces in theory 1.66E+15 unique libsodium variants. Yet, it is unpractical to store and execute this large number of variants. Thus, we sample the pool of possible variants to evaluate our generated variants. First, for each of the 256 modules, we rank each module variant with respect to the number of lines changed in the final WebAssembly textual format. Then, to produce the i-th library variant, we combine the i-th variant for each module of libsodium, in order to produce maximally diversified library variants first. If a module has less than i variants, we use the original, non-diversified module. According to Table II, the maximum number of unique variants for a single module is 79 (`codecs` module). Thus, we sample 79 unique libsodium variants, ordered by the amount of diversification (the first variant contains the most changes, and so on). For each variant we execute the complete test suite to validate its correctness. All test cases successfully pass for all diversified library binaries.

Dynamic evaluation of libsodium variants. We compare the dynamic behaviour of the original libsodium and the 79 library variants. Figure 3 illustrates the distribution of dt_dyn of all collected traces for each libsodium test. The dt_dyn distance is calculated between each diversified trace

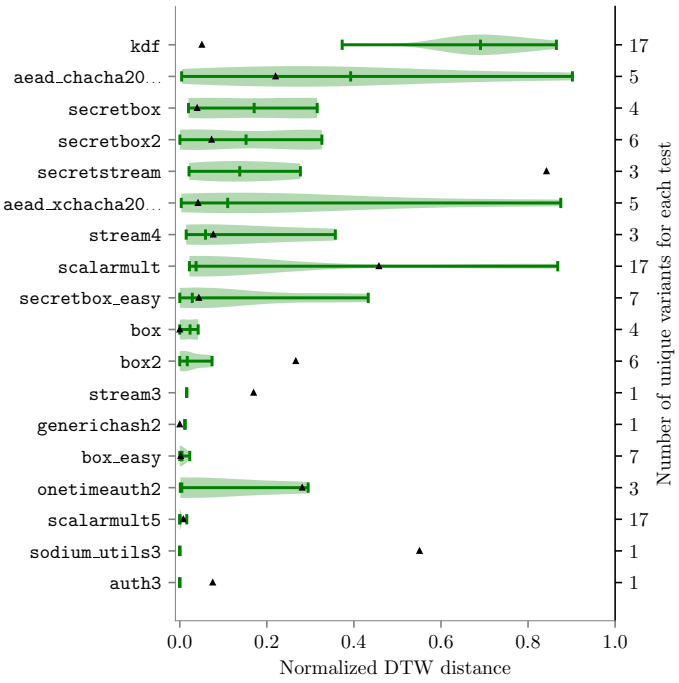

Fig. 3: Distribution of normalized dt_dyn distances over the set of libsodium variants covered by each test. The left Y axis lists the name of each test. The number of unique variants used per test is listed on the right Y axis. The black triangles point to the dt_dyn distance between two different stack operation traces of the original test with different random seeds.

and the corresponding original trace for the same test. Each horizontal bar gives the distribution of dt_dyn over the 79 diversified libraries per test. The black triangles show the dt_dyn distance between two different executions of the same test with different random seeds. They serve as a baseline to compare the artificial diversity introduced by CROW, against the natural trace diversity that appears because of random number generation.

For $18/19$ tests, we observe that CROW's diversified modules produce a different trace than the original. The wider violin plots that reach the right-hand side of the figure include variants that significantly diversify the test execution. We observe that $4/18$ tests stand out as they include variants with at least 0.8 normalized dt_dyn distance. For $6/18$ tests, there is a medium trace diversity as their dt_dyn distributions lie in the mid/left side of the plot. For the rest $8/18$ tests we observe a significantly smaller dt_dyn distance.

This means that, in the context of this cryptographic library, CROW is able to find variants that have a huge impact on the dynamic stack behaviour of the program. Meanwhile, some other replacements can have only a marginal impact during the operation of the program. One factor that can affect this is the "centrality" of the code that is being replaced. Diversified code that is called often, potentially inside loops, will have a greater impact on the stack trace of a program compared to code that is only called, for example, only during the initialization of the program.

When we compare the trace diversity against the diversity due to pseudo-number generation (black triangles in Figure 3), we observe that: for $2/18$ tests CROW trace diversification is always larger than the one due to random number generation, for $11/18$ tests there exist some variants that exhibit larger trace diversification than random number generation and for $5/18$ tests CROW trace diversification is always smaller than the one due to random number generation.

Answer to RQ3

We have successfully applied CROW to libsodium, one of the leading WebAssembly cryptography libraries. We have shown that CROW is able to create statically different variants of this real-world library, all of which being distributable to users. Our original experiments to measure the trace diversity of libsodium have proven that the generated variants exhibit significantly different execution traces compared to the original non-diversified libsodium binary. The take-away of this experiment is that CROW works on complex code.

VI. Threats to Validity

Internal: The timeout in the exploration stage is a determinant factor to generate unique variants. It is required to bound the experimental time. If the timeout is increased, the number of variants and unique variants might increase.

External: The 303 programs in our Rosetta corpus may not reflect the constructs used in the WebAssembly programs in the wild. Yet our experiment on libsodium shows that the results on the Rosetta corpus hold on real code. To increase external validity, we hope to see more benchmarks of WebAssembly programs published by the research community.

Scale: We measure behavioral diversity with DTW. We are aware that this behavioral diversity metric does not scale infinitely. To make comparisons between large execution traces, it may be necessary to use a more scalable metric. To mitigate this scale problem in future work, one option is to compare software traces using entropy analysis, as proposed by Miranskyy et al. [33].

VII. Related Work

Program diversification approaches can be applied at different stages of the development pipeline.

Static diversification: This kind of diversification consists in synthesizing, building and distributing different, functionally equivalent, binaries to end users. This aims at increasing the complexity and applicability of an attack against a large population of users [12]. Jackson et al. [24] argue that the compiler can be placed at the heart of the solution for software diversification; they propose the use of multiple semantic-preserving transformations to implement massive-scale software diversity in which each user gets their own diversified variant. Dealing with code-reuse attacks, Homescu et al. [23] propose inserting NOP instruction directly in LLVM IR to generate a variant with different code layout at each compilation. In this area, Coppens et al. [13] use compiler transformations to iteratively diversify software. The aim of their work is to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach, continuously applies a random

selection of predefined transformations using a binary diffing tool as feedback. A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them. Our work can be extended to address this issue, providing a synthesizing solution which is more general than specific transformations.

The work closest to ours is that by Jacob et al. [25]. These authors propose the use of a "superdiversification" technique, inspired by superoptimization [32], to synthesize individualized versions of programs. In the work of Massalin, a superoptimizer aims to synthesize the shortest instruction sequence that is equivalent to the original given sequence. On the contrary, the tool developed by Jacob et al. does not output only the shortest instruction sequence, but any sequences that implement the input function. This work focuses on a specific subset of X86 instructions. Meanwhile, our approach works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures. Specifically, we apply our tool on WebAssembly, something not possible with the X86-specific approach of that paper.

Runtime diversification: Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces that a side-channel attack requires to succeed. Amarilli et al. [3] are the first to propose generation of code variants against side-channel attacks. Agosta et al. [1] and Crane et al. [15] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software, while Couroussé et al. [14] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. CROW focuses on static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a dynamic approach. Since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW.

VIII. Conclusion

Security has been a major driver for the design of WebAssembly. Diversification is one additional protection mechanism that has been not yet realized for it. In this paper, we have presented CROW, the first code diversification approach for WebAssembly. We have shown that CROW is able to generate variants for a large variety of programs, including a real-world cryptographic library. Our original experiments have comprehensively assessed the generated diversity: we have shown that CROW generates diversity both among the binary code variants as well as in the execution traces collected when executing the variants. Also, we have successfully observed diverse execution traces for the considered cryptographic library, which can protect it against a range of side channel attacks.

Future work includes increasing the number of unique variants that are generated, by working on block replacement overlapping detection. Also, the exploration stage and the identification of code replacements is a highly parallelizable process, this would increase diversification performance in order to meet the demands of the internet scale.

References

[1] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1320–1333, 2015.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ch. 1, pp. 28–31.

[3] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can code polymorphism limit information leakage?" in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2011, pp. 1–21.

[4] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[5] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–26, 2015.

[6] D. J. Bernstein, "The ChaCha family of stream ciphers," 2008. [Online]. Available: http://cr.yp.to/chacha.html

[7] ——, "The Salsa20 family of stream ciphers," in *New stream cipher designs*. Springer, 2008, pp. 84–97.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[9] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 292–302.

[10] J. Cabrera Arteaga, M. Monperrus, and B. Baudry, "Scalable comparison of javascript V8 bytecode traces," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–31.

[11] D. Chen and W3C group, "WebAssembly documentation: Security," W3C, Accessed: 18 June 2020. [Online]. Available: https://webassembly.org/docs/security/

[12] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[13] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–26, 2013.

[14] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *IFIP International Conference on Information Security Theory and Practice*. Springer, 2016, pp. 136–152.

[15] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[16] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving target defense*. Springer, 2011, pp. 99–108.

[17] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[18] F. Denis, "The Sodium cryptography library," Jun 2013. [Online]. Available: https://download.libsodium.org/doc/

[19] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[20] R. Gurdeep Singh and C. Scholliers, "WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019, 2019, pp. 27–36.

[21] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 185–200.

[22] M. Hamburg, H. de Valance, I. Lovecruft, and T. Arcieri, "The ristretto group," 2017.

[23] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–11.

[24] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*. Springer, 2011, pp. 77–98.

[25] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*. Springer, 2008, pp. 100–120.

[26] S. Josefsson, "The Base16, Base32, and Base64 data encodings," Internet Requests for Comments, RFC Editor, RFC 4648, October 2006, http://www.rfc-editor.org/rfc/rfc4648.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4648.txt

[27] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," Internet Requests for Comments, RFC Editor, RFC 2898, September 2000, http://www.rfc-editor.org/rfc/rfc2898.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2898.txt

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[29] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 216–226.

[30] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[31] M. D. A. Maia, V. Sobreira, K. R. Paixão, R. A. D. Amo, and I. R. Silva, "Using a sequence alignment algorithm to identify specific and common code from execution traces," in *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, 2008, pp. 6–10.

[32] H. Massalin, "Superoptimizer— A Look at the Smallest Program," *ACM SIGPLAN Notices*, vol. 22, no. 10, pp. 122–126, 10 1987.

[33] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza, "Using entropy measures for comparison of software traces," *Information Sciences*, vol. 203, pp. 59–72, oct 2012.

[34] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009.

[35] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 297–310, Mar. 2016.

[36] A. Rossberg, "WebAssembly Core Specification," W3C, Tech. Rep., Dec. 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[37] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity," in *NDSS*, 2017.

[38] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[39] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[40] M. Taguinod, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward a moving target defense for web applications," in *2015 IEEE International Conference on Information Reuse and Integration*. IEEE, 2015, pp. 510–517.

[41] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2001, pp. 193–202.

[42] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense*, 2014, pp. 31–40.